Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Klemen Koželj

Postavitev skalabilne arhitekture za napovedovanje razpoložljivosti postaj sistema BicikeLj

diplomsko delo

visokošolski strokovni študijski program prve stopnje računalništvo in informatika

Mentor: viš. pred. dr Aleksander Sadikov

Ljubljana, 2017

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Študent naj postavi računalniško gručo, ki bo omogočala horizontalno skalabilnost obdelave podatkov. Študent naj preuči in nato gručo zastavi iz dveh tehnologij in sicer Apache Hadoop za shranjevanje podatkov in Apache Spark za obdelavo le-teh. Iz javnih API-jev naj pridobi podatke o uporabi postaj sistema BicikeLj ter vremenske podatke. Zbrane podatke in računalniško gručo naj uporabi za napovedovanje polnosti postaj sistema BicikeLj z uporabo metod podatkovnega rudarjenja.

Za podporo med svojim dosedanjim šolanjem se zahvaljujem svoji mami, stari mami Anici in staremu očetu Štefanu.

Prav tako se zahvaljujem mentorju, viš. pred. dr. Aleksandru Sadikovu, za nasvete in vodenje med izdelavo.

Moji mami.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
API	application programming interface	vmesnik za uporabvno programiranje
NPM	node package manager	nodov upravljavec paketov
OWM	open weather maps	spletni portal z vremenskimi podatki
JSON	Java Script object notation	objekt Java Script
IP	internet protocol	internetni protokol
HDFS	Hadoop distibuted file system	Hadoopov porazdeljen datotečni sistem
YARN	yet another resource manager	Hadoopov upravljavec virov
VPN	virtual private netowrk	navidezna zasebna mreža
DHCP	dynamic host configuration protocol)	protokol za avtomatsko dodeljevanje IP-naslovov
SSH	secure shell protocol	protokol varovane lupine
FTP	file transmission protocol	protokol za pošiljanje datotek

XML	extensible markup language	razširjen znakovni jezik
RAM	random access memory	bralno-pisalni pomnilnik
RDD	resilient distributed drive	odporen distribuiran pomnilnik
TCP	transmission control protocol	protokol za nadzorovan prenos podatkov
k-NN	k nearest neighbours	k najbližjih sosedov
RMSE	root mean square error	koren povprečne kvadratne napake
MAE	mean absolute error	absolutna povprečna napaka

Povzetek

Naslov: Postavitev skalabilne arhitekture za napovedovanje razpoložljivosti postaj sistema BicikeLj

Dandanes smo priča pravi digitalni revoluciji v vseh panogah industrije, ki se ukvarjajo z mobilnostjo ali s transportom. Kot zgled lahko izpostavimo ameriški start up Uber, ki je v nekaj letih po vsem svetu digitaliziral mobilnost. Njegov celotni poslovni model sloni na podatkovnem rudarjenju, saj z metodami poskuša odgovarjati na vprašanje: "Od kod in kam bodo ljudje potovali?"ter za to potrebovali njihove storitve. Bolje kot lahko predvidijo gibanje populacije skozi neko področje, bolj uspešno in optimizirano je lahko njihovo poslovanje na trgu. V tej diplomski nalogi bomo sprva zbrali podatke o povpraševanju ljudi po mobilnosti v določenih predelih mesta in nato postavili učinkovit ter horizontalno razširljiv sistem, ki bo omogočal njihovo shranjevanje in obdelavo. Najbolj primeren za tovrstni projekt je ljubljanski sistem za izposojo koles, poimenovan BicikeLj. Vprašanje, na katerega bomo iskali odgovor, se glasi, kako zasedene bodo postaje BicikeLj ob izbranem času v prihodnosti in kako natančno znamo to napovedati iz preteklih podatkov.

Ključne besede: apache hadoop, apache spark, podatkovno rudarjenje.

Abstract

Title: Scalable architecture for availability prediction of BicikeLj stations

Recently, we are witnessing a true digital revolution in every sector of the industry, which is involved in mobility and transportation. As an example, we can expose American start-up Uber, which has drastically digitalized mobility in the recent years. Their business strategy is data driven; with the data mining methodologies they are trying to answer a question "From and where people will travel?", and for that they use their services. The better they can predict the movement of people through the area, the more successful and optimised their actions on the market are. In this bachelor thesis, first we will collect the data of movement of people through the city of Ljubljana, then we will form horizontally scalable system, which will enable us persistent storage and processing of the collected data. The most suitable system for this kind of project is bike sharing system located in Ljubljana, which operates under the brand BicikeLj. The question, on which we will try to answer, is how empty or full BicikeLj stations will be in a certain time in the future, and how precisely we can predict that based on the previous data.

Keywords: apache hadoop, apache spark, data mining.

Poglavje 1

Uvod

Vsi sistemi, ki svojim strankam ponujajo storitve mobilnosti, se srečujejo s podobnimi težavami. Za uspešno poslovanje sistema morajo biti vedno na razpolago. Če stranka potrebuje prevoz s točke A na točko B in ponudnik tega ne more zagotoviti, to zamaje same temelje poslovanja. Ljubljanski sistem izposoje koles BicikeLj ima dandanes na področju Mestne občine Ljubljana, kjer tudi aktivno posluje, 38 aktivnih postajališč, kjer si lahko uporabniki izposodijo ali pa pustijo izposojeno kolo. Če je postaja popolnoma prazna ali popolnoma polna, si uporabnik ne more izposoditi oziroma vrniti kolesa, kar nakazuje na neučinkovito stanje sistema, ki se kasneje lahko odraža v nezadovoljnih strankah. Omenjeno težavo večina podjetij iz panoge rešuje ročno, kar pomeni, da s kombiji razvažajo kolesa s polnih na prazna postajališča in obratno ter tako skrbijo za uravnoteženo stanje.

Če bi ponudniki teh storitev lahko uspešno napovedovali povpraševanje po svojih storitvah ne nekem področju, bi se lahko vnaprej pripravili tako, da do disharmonije sploh ne bi prišlo. Tako bi lahko na primer uporabili napredne koncepte, kot je recimo vpreženje množice (angleško crowdsoursing).

Če podjetje z dovolj visoko gotovostjo ve, da se bo neko postajališče koles v naslednjih nekaj urah popolnoma zapolnilo in drugo bližnje postajališče popolnoma izpraznilo, bi lahko uporabnikom recimo ponudilo neke ugodnosti, če kolo vrnejo na postajališču, kjer lahko pričakujemo deficit koles. Če bi bila nagrada uporabnikom dovolj mamljiva in bi dejansko zato pustili kolo na želeni postaji, podjetju ne bi bilo treba ročno prestavljati kolesa, kar bi lahko zanj pomenilo zmanjšanje stroškov poslovanja. Vedeti pa moramo, da je opisani sistem zelo občutljiv, saj če podjetje ponudi preveliko nagrado, začne proizvajati izgubo, saj lahko ročno prestavljanje koles postane zanj ugodnejše.

Kot vidimo, vsa uspešnost sistema sloni na dejstvu, kako dobro lahko napovemo polnost specifičnega postajališča v prihodnosti.

Za rešitev problema sprva potrebujemo podatke, iz katerih bomo kasneje izluščili odgovore na zastavljena vprašanja. Na srečo francosko podjetje JCDecaux, ki je implementiralo in postavilo ljubljanski sistem BicikeLj, ponuja odprte in vsem dostopne API-je, kjer lahko preprosto v živo zajemamo informacije o trenutnem stanju sistema. Sklepamo lahko, da je uporaba koles odvisna tudi od vremenskih pogojev, ki jih lahko dobimo s spletnega portala OWP, ki ponuja številne API-je, povezane z vremenom.

V prvem delu diplomskega dela je opisana kombinacija tehnologij, s katero sem poskusil zagotoviti kar se da zanesljivo in natančno zbiranje podatkov. Zavedal sem se, da mi bodo lepo strukturirani podatki, zajeti v natančnih časovnih intervalih, olajšali prihajajoče delo.

Sami podatki so nični, če nimamo postavljenega sistema, kjer jih lahko učinkovito hranimo in obdelujemo. Za hrambo in obdelavo večjih podatkovnih zbirk pogosto ne zadostuje en sam računalnik, zato zato sem se odločil za postavitev računalniške gruče s tehnologijama Hadoop in Spark, ki ju razvija programska fundacija Apache.

V drugem delu diplomske naloge sem predstavil prednosti računalniške gruče in arhitekturo obeh uporabljenih tehnologij. Opisal sem tudi postopek postavitve in uporabe celotnega sistema.

V zadnjem delu naloge sem iskal odgovor na postavljeno vprašanje: "Kako zasedene bodo postaje BicikeLj v nekem prihodnjem času?Ž metodami podatkovnega rudarjenja sem iskal odgovor v zbranih podatkih in kot rezultat zgradil napovedovalni model. Zadnje dejanje praktičnega dela je bila objektivna ocenitev praktične uporabne vrednosti modela.

Uporabljal sem programski jezik Python v kombinaciji s Sparkovimi API-ji in nekaj knjižnicami, ki olajšajo delo s podatki ali pa že imajo implementirane potrebne algoritme.

Poglavje 2

Zbiranje podatkov

Podatke sem zbiral in shranjeval od 15. 4. 2016 do 13. 7. 2016. Prebiral sem jih iz javnih API-jev podjetja JCDecaux [6] in s portala OWM [7]. Podatke sem prejemal v JSON-formatu. Podatke o polnosti postaj sem zajemal vsako minuto, vremenske podatke pa vsake 3 ure, saj so se po tem intervalu posodabljale tudi informacije na API-jih. Da sem podatke zajemal kar se da točno, sem uporabil Linuxovo orodje, imenovano Cron. Cron je klical program, napisan v ogrodju NodeJS, ki je nato prebral podatke iz API-jev in jih shranil v podatkovno bazo MongoDB. Program je bil prav tako povezan s Slackom, tj. orodjem za neposredno komunikacijo v realnem času. Na Slack je program sporočal vse posebnosti in napake v izvajanju. Tako ni bilo mogoče, da bi se zajemanje podatkov iz kakršne koli napake ustavilo in o tem ne bi bilo obveščen.

2.1 Cron

Cron [20] je Linuxovo orodje, s pomočjo katerega lahko kreiramo urnik, po katerem nato operacijski sistem izvaja želene operacije. Zelo specifično lahko definiramo interval izvajanja, zato je Cron idealen za izvrševanje časovno ponovljivih nalog.

Sama sintaksa Crona je zelo enostavna. Če v ukazno lupino vpišemo ukaz "crontab -e", se nam v našem prevzetem urejevalniku besedil odpre datoteka, ki je v direktoriju /var/spool/cron/$USER. To je Cronova privzeta konfiguracijska datoteka, v katero vpišemo urnik, po katerem se nato izvajajo želeni ukazi. V omenjeni datoteki je vsaka vrstica namenjena svojemu dogodku. Sprva zapišemo interval izvajanja, nato ukaz. Interval izvajanje je predstavljen s petimi zvezdicami, števili ali enostavnim računom. Zvezdica simbolizira katerokoli numerično vrednost, število jo enolično določa, enačba se izvrši, ko je njena vrednost enaka nič. Prvi simbol predstavlja minuto, drugi uro, tretji dan v mesecu, četrti mesec ter poslednji dan v tednu. Če v Bash vpišemo ukaz črontab -l", se nam v pregled izpiše vsebina Cronove konfiguracijske datoteke.

Listing 2.1: Izpis Cronove konfiguracijske datoteke.

```
* * * * * node /home/$USER/DataCollector/index.js bicikelj
15 */3 * * * node /home/$USER/DataCollector/index.js weather
```

2.2 NodeJS

Program, ki ga je klical Cron, je bil napisan v programskem jeziku Java Script, natančneje v ogrodju NodeJS [8]. Ogrodje NodeJS nam omogoča, da Javo Script, ki se je včasih primarno uporabljala na čelnih sistemih, prenesemo tudi na zaledne. Temelji na Googlovem pogonu V8, ki prevaja skriptni jezik, kot je JavaScript, v C++. Sam sistem izvajanja NodeJS je podatkovno gnan in uporablja le eno procesorsko nit. Zaradi teh karakteristik spodbuja k popolnoma asinhronemu programiranju. Skupaj z NodeJS pa dobimo tudi NPM. NPM je Nodov privzeti upravljavec modulov, s katerim lahko dostopamo do trenutno največje zbirke knjižnic na svetu.

- **Mongoose**: [12] modul, ki nam olajša delo in povezovanje s podatkovno bazo MongoDB. S pomočjo Mongoosa lahko vnaprej definiramo sheme za podatke oziroma JSON-objekte, ki jih imamo namen shraniti ali prebrati iz podatkovne baze.

- **Async**: [11] uporabljamo za nadzor izvajanja. Jasno lahko definiramo, katere operacije se naj izvedejo sinhrono ali asinhrono. Tako lahko recimo enostavno sinhrono iteriramo skozi elemente seznama in nad vsakim elementom vršimo asinhrono operacijo.

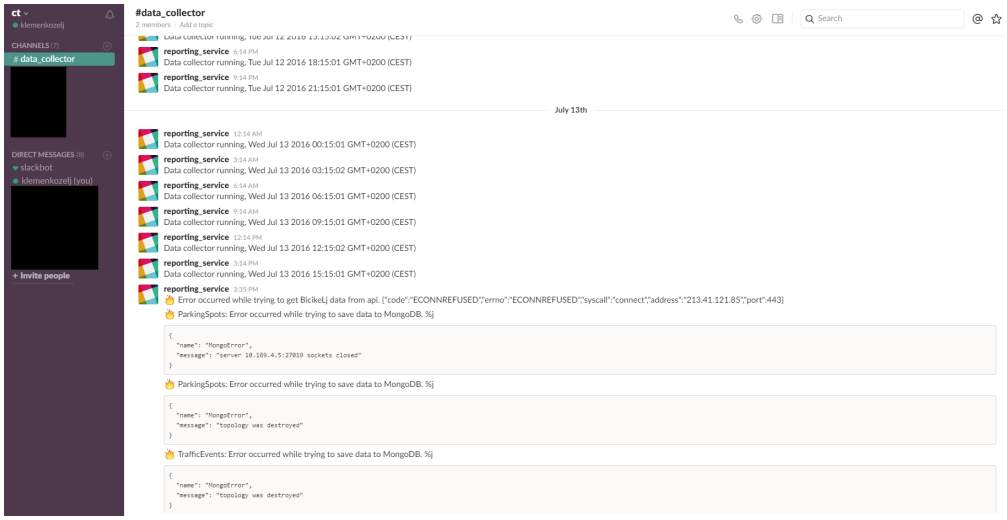
- **Request**: [14] nam poenostavi delo s HTTP-zahtevki. Modul pripomore k berljivost kode in omogoči lažje posredovanje v primeru napak.

- **Winston**: [15] je modul za beleženje dogodkov, do katerih pride med izvajanjem programa. Vsak zapis lahko razdelimo v kategorije po pomembnosti in na kateri kanal naj se ta pošlje.

- **Winston-Slackbotuser**: [16] dodatek, ki dopolnjuje Winston. Z njim lahko pošiljamo zapise o dogodkih na Slack, kjer ustvarimo virtualnega robota, ki ga Slackbotuser uporablja za objavljanje sporočil.

2.3 Slack

Slack [9] je komunikacijsko orodje, ki se uporablja za komuniciranje v realnem času. Izredno je priljubljen med inženirji zaradi odlične kompatibilnosti z ostalimi tehnologijami, ki se uporabljajo pri razvoju programske opreme. Tudi sam sem povezal program za pridobivanje podatkov na poseben kanal Slack. Kot vidimo s spodnjega zaslonskega posnetka2.1, se je moj virtualni robot javil vsake tri ure, tako ni bilo dvoma o tem, da opravlja svoje delo. Če je prišlo do kakršnekoli napake med njegovim izvajanjem, pa je to tudi nemudoma sporočil in v priponko v JSON-obliki dodal informacije, ki bi jih lahko uporabili pri odpravljanju problema.



Slika 2.1: Zaslonski posnetek kanala Slack, ki ga je uporabljal program za pridobivanje podatkov.

2.4 MongoDB

Je trenutno najbolj popularna ne relacijska podatkovna baza [10]. Popularnost med razvijalci opravičuje z odprtokodnostjo, odlično arhitekturo, kar nam omogoči horizontalno skalabilnost, in zelo močnim poizvedbenim jezikom, s katerim lahko pridobimo praktično kakršnekoli informacije iz shranjenih podatkov. Samo povezovanje na bazo opravimo z Mongo URI-jem, ki vsebuje vse potrebne informacije, kot so IP, vrata, uporabniško ime in geslo. Mongo vsakemu zapisanemu objektu doda 12-bitno heksadecimalno število. To število je unikatno in tako enolično referencira objekt v bazi.

Podatke hrani v JSON-formatu, kar nam je omogočalo v enakem formatu prebrane podatke iz API-jev enostavno shraniti v podatkovno bazo. V našem primeru smo podatkovno bazo Mongo uporabljali le za začasno hrambo podatkov, saj smo jih kasneje izvozili v Hadoopov HDFS.

2.5 Python in pomembnejše knjižnice

Python [21] je splošno razširjen visokonivojski programski jezik, ki se uporablja na različnih področjih računalništva, leta 1991 ga je razvil Guido von Rossum. Python se interpretira, kar pomeni, da se predhodno ne prevaja v strojni jezik. Glavna filozofija Pythona je berljivost kode, kar omogoča z izjemno lahko sintakso. Ker ponuja enostavnost, omogoča hitro dosego želenega efekta, brez odvečne kode je posebej priljubljen v akademskem svetu.

Python sem uporabljal za podatkovno rudarjenje nad izbranimi podatki v kombinacijami s Sparkovimi API-ji in s knjižnicami Numpy, Pands, Matplotlib, Sklearn, Folium ter Pymongo.

- **Numpy**: [22] nam olajša delo z večdimenzionalnimi podatkovnimi strukturami. Tako lahko laže indeksiramo elemente, jih filtriramo ...
- **Pandas**: [23] podobna knjižnica kot Numpy, lahko ju uporabljamo v kombinaciji. Nad večdimenzionalne tabele vpelje večjo abstrakcijo, ki jo imenujemo DataFrame, tako nam še dodatno olajša delo s podatki.
- **Matplotlib**: [24] knjižnica za izrisovanje grafov, najlažje jo uporabljamo z vektorji Numpy.
- **Sklearn**: [13] Vsebuje implementacije velike večine algoritmov, ki jih uporabljamo v podatkovnem rudarjenju. Prav tako vsebuje funkcije za urejanje podatkov.
- **Pymongo**: [26] knjižnica za povezovanje in delo s podatkovno bazo MongoDB.
- **Folium**: [25] orodje za ustvarjanje interaktivnih zemljevidov.

Poglavje 3

Računalniška gruča

V kolikor se ozremo nekaj let v preteklost, je bila tipična arhitektura nekega splošnega računalniškega sistema sestavljena iz dveh komponent. Prva komponenta je imela nalogo procesirati podatke, druga komponenta pa je bila zadolžena za njihovo shranjevanje. Komponenti sta bili odvisne druga od druge in sta skupaj tvorili delujočo celoto. Splošno smo uporabljali dve napravi, prvo za procesiranje in drugo za hrambo podatkov. V primeru manjših sistemov pa je oboje lahko shajalo celo na isti napravi.

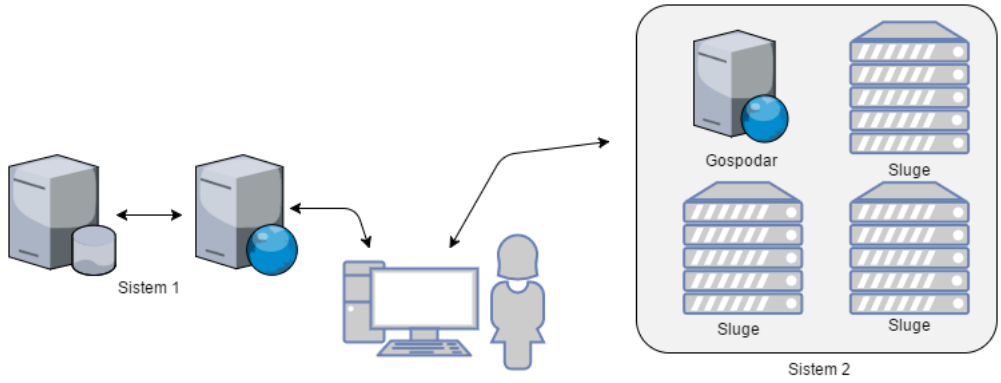
Tak pristop deluje in je zadovoljiv, vendar ima svoje omejitve. Kot prvo omejitev bi lahko omenili samo računsko moč, saj smo omejeni na eno napravo za vsako nalogo. Posledično to pomeni, da sam sistem lahko skalira samo vertikalno. Druga omejitev je to, da vse naše zaupanje sloni prav tako na vsaki posamični napravi. Če pride do kakršne koli okvare, naš sistem kot celota pade v vodo. V sodobnem poslovnem svetu pa si tega enostavno ne moremo privoščiti.

Rešitev je računalniška gruča. Omenjena gruča oziroma (angleško computer cluster) je sistem, v katerega je povezanih več računalnikov. Računalniki so navadno povezani v lokalnem omrežju in navzven tvorijo končno celoto.

Vsako vozlišče gruče je svoja entiteta, vendar skupaj kot celota rešujejo dani problem. Z združitvijo dobimo večjo računsko moč pa tudi bolj tolerantni smo do okvar in napak.

Če poljubno vozlišče zaradi kakršnega koli razloga preneha delovati, izgubimo nekaj virov, vendar to ne podre celotnega sistema. Računalniške gruče ne smemo zamenjevati s tako imenovanim mrežnim računanjem. Sistema sta si med seboj podobna, a pri računalniški gruči vsa vozlišča rešujejo enak problem oziroma izvajajo enake naloge, pri mrežnem računanju pa ne.

Na spodnji sliki 3.1 lahko na levi vidimo shemo navadnega sistema, na desni strani pa sistem gruče računalnikov. Kot pomanjkljivost računalniške gruče pa moramo omeniti same stroške dodatne strojne opreme in povečano porabo električne energije.



Slika 3.1: Shema računalniškega sistema z gručo in brez nje.

Oba sistema, Hadoop in Spark, ki sem ju uporabil, sledita enakemu arhitekturnemu konceptu, poimenovanem gospodar-suženj (angleško master-slave). Pri takem konceptu ima eno izmed vozlišč vlogo gospodarja, ostala pa so sužnji.

Vozlišče z vlogo gospodarja razdeljuje naloge, razpolaga z viri, preverja stanja vozlišč, ki igrajo vlogo sužnja, in podobno. Vozlišča z vlogo sužnja izvajajo navodila gospodarja. Na gospodarjevem vozlišču je navadno tudi čelni sistem, na katerega se povezujemo in tako upravljamo s celotno gručo.

Če vozlišče, ki deluje kot gospodar, odpove, imamo tudi sekundarnega gospodarja, lahko pa tudi sam sistem izglasuje novega. Če pa odpove vozlišče z vlogo sužnja, se njegove naloge prerazporedijo na druga suženjska vozlišča. Če ne izgubimo prevelikega števila suženjskih vozlišč, sistem ne preneha delovati, vsaka izguba pa seveda pomeni večjo obremenitev ostalih vozlišč. Dodatne računalnike lahko poljubno dodajamo in prav tako odvzemamo med samim delovanjem gruče.

3.1 Hadoop

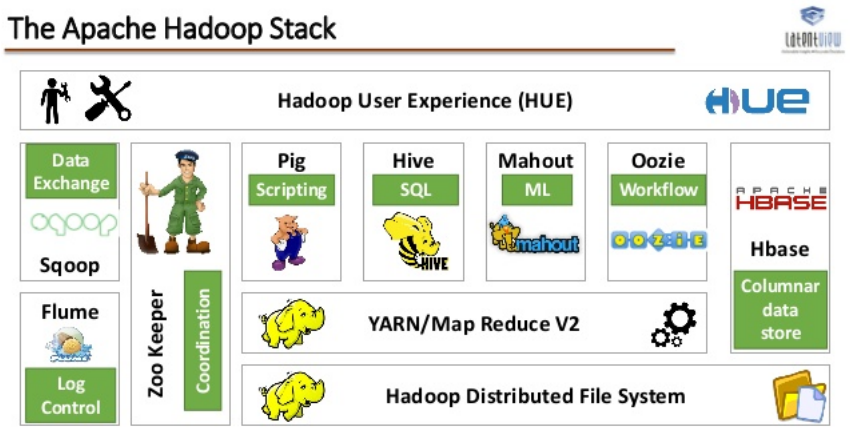
Računalniški gigant Google je leta 2004 za obdelavo velikih podatkovnih zbirk na gruči začel uporabljati algoritem MapReduce. Algoritem razdeli problem v manjše podprobleme in vsak podproblem dodeli svoji napravi. Tako paralelno obdelamo podatke, na koncu pa rešitve združimo v končni rezultat.

Doug Cutting [28] je opazil potencial algoritma in se lotil projekta Hadoop ter tako razvil odprtokodno okolje za hrambo in procesiranje podatkov na gruči računalnikov. Napisan je v programskem jeziku Java, kar nam omogoči, da ga uporabljamo na katerekoli platformi.

Največjo gručo Hadoop naj bi imeli pri podjetju Yahoo. Gruča naj bi imela 40.000 računalnikov, ki imajo skupaj več kot 100.000 procesorjev. Gruča naj bi shranjevala več kot 455 petabajtov podatkov [30].

3.1.1 Hadoopov sklad

Projekt Hadoop [2] je trenutno v lasti programske fundacije Apache, ki ga tudi aktivno razvija skupaj z ostalimi sorodnimi tehnologijami. Na spodnji sliki 3.2 lahko vidimo prikazan sklad vseh Hadoopovih tehnologij. V okviru diplomske naloge sem uporabljal le HDFS za dolgoročno hrambo podatkov.



Slika 3.2: Pregled vseh Hadoopovih tehnologij.

HDFS predstavlja jedro Hadoopa, na zgornji sliki 3.2 ga lahko opazimo na skrajnem dnu sklada, je Hadoopov porazdeljen datotečni sistem (angleško Hadoop Distributed File System). Ponuja nam možnost shranjevanja podatkov. Podatki so razpršeni na mnogih napravah, ki so del gruče. Podatke, ki jih pošljemo na HDFS, Hadoop sprva razdeli v manjše bloke velikosti 128 MB, lahko tudi 64 MB. Nato podatkovne bloke razpošlje na vse računalnike v gruči, upošteva tudi faktor razpršitve, ki ga sami definiramo. Večji kot je faktor razpršitve, bolj varni so podatki, saj so prepisani na večjih mestih v gruči, seveda pa moramo v obzir vzeti tudi to, da nam to prinese veliko podatkovno redundanco.

Sistem je zasnovan tako, da ga lahko poganjamo na povsem običajni strojni opremi in ne potrebujemo izjemno zmogljivih strežnikov.

Nad HDFS-jem leži MapReduce, Googlov algoritem za paralelno procesiranje podatkov, ki smo ga že omenili zgoraj. Poleg opazimo še kratico YARN [2], ki simbolizira "Yet Another Resource Negotiator". YARN je Hadoopov privzeti sistem za upravljanje z gručo. Z njim lahko definiramo urnik procesov, ki jih bomo izvedli na gruči, in vire, ki jih dodelimo vsakemu posamičnemu procesu. Tako lahko bolj optimalno upravljamo in razpolagamo z gručo.

3.1.2 Arhitektura Hadoopa

Namenode

Predstavlja osrednji proces HDFS-ja in se izvaja na računalniku, ki smo ga izbrali kot gospodarja. Sam ničesar ne shranjuje, temveč operira z metapodatki. V drevesni podatkovni strukturi hrani informacije, kje v gruči je kaj shranjeno. Prav tako pa izvaja oziroma nadzira vse premike, brisanja, kopiranja in druge operacije nad HDFS-jem. Zaradi odzivnosti in hitrosti so metapodatki hranjeni v pomnilniku, seveda pa moramo metapodatke ob primernem času zapisati še na disk ter jih tako trajnostno shraniti. Namenode operira z dvema datotekama, poimenovanima "fsimage" in "edit logs".

V prvi datoteki z imenom "fsimage" je shranjeno datotečno stanje HDFS-ja pri njegovem zagonu. V drugo datoteko "edit logs" pa zapisujemo vse storjene spremembe na sistemu HDFS. Ob ugasnitvi Namenoda je vsebina iz "edit logs" kopirana na "fsimage", s čimer dobimo najnovejše stanje za naslednji zagon HDFS-gruče. V velikih HDFS-sistemih, ki so uporabljeni v produkciji, kar pomeni, da jih ne ugašamo pogosto, kaj kmalu pride do tega, da je datoteka "edit logs" enormno velika, kar nas pripelje do treh večjih problemov. Prvič, velike datoteke so težje obvladljive kot manjše. Ob morebitni potrebi po ugasnitvi in ponovnem zagonu HDFS-ja bi nam to vzelo ogromno časa, saj moramo posodobiti "fsimage" z veliko spremembami. Zadnji in najslabši scenarij, ki se nam lahko pripeti, je, da če se sistem popolnoma podre in vsebina iz "edit logs"ni bila kopirana, izgubimo podatke. Namenode štejemo tudi kot Ahilovo peto HDFS-ja. Če počepne, je ves sistem neodziven, saj nimamo vstopne oziroma izstopne točke v gručo.

Secondary Namenode

Ko preberemo ime tega procesa, bi laično pomislili, da gre za rezervni Namenode, ki se aktivira, če bi zaradi kakršnegakoli razloga zgoraj omenjeni Namenode prenehal delovati. V resnici ima Secondary Namenode drugačen namen, se pa prav tako izvaja na napravi, ki ima pravice gospodarja.

Namenode v rednih časovnih intervalih pošilja spremembe, ki zapisane v datoteki "edit logs" Secondary Namenodu, ta pa z njimi posodablja "fsimage" datoteko. Nemudoma, ko Secondary Namenode posodobi "fsimage" z najnovejšimi prejetimi podatki, jo pošlje nazaj k Namenodu. Kot vidimo, je goli namen Secondary Namenoda posodabljanje "fsimage" datoteke, s čimer skrbi za bolj pogosto shranjevanje stanja celotnega sistema.

Datanode

Izvaja se na vseh računalnikih v gruči, ki smo jih izbrali kot sužnje. Seveda pa lahko na računalniku, ki opravlja delo gospodarja, vzporedno teče tudi proces Datanode. Ob zagonu se Datanode poveže z Namenode in nato po navodilih, ki jih prejema, obdeluje, shranjuje ali briše lokalno shranjene podatke, ki so del HDFS-ja.

3.1.3 Konfiguracija Hadoopa

Omrežje

Vsi računalniki v gruči morajo biti prisotni v enakem omrežju. Lahko uporabimo tudi VPN, vendar to seveda negativno vpliva na končni performans. Dobra praksa je, da računalnikom dodelimo statične IP-naslove tako, da v vsakem trenutku vemo, preko katerega IP-naslova se lahko povežemo do katerega računalnika. Statični IP lahko dodelimo na DHCP-strežniku ali pa na samem računalniku [27]. Najbolje, da to storimo tako, da se bodo nastavitve obdržale tudi po ponovnem zagonu naprave, konfiguracijo zapišemo v datoteko "/etc/network/interfaces". Z ukazoma "ifdown" in "ifup" pa le ugasnemo ter prižgemo mrežni vmesnik tako, da nemudoma dobimo želene nastavitve.

Naslednja naloga je, da vse IP-naslove računalnikov, ki so v gruči, zapišemo v datoteko "/etc/hosts" poznanih strežnikov in jim dodelimo vzdevke.

Listing 3.1: Nastavitev mrežnega vmesnika eth0 s statičnim IP-naslovom.

```
auto eth0
iface eth0 inet static
address 10.80.17.100
netmask 255.255.255.0
gateway 10.80.17.254
```

Listing 3.2: Ureditev Linuxove "/etc/hosts"datoteke.

```
10.80.17.101 Master
10.80.17.102 Slave1
10.80.17.103 Slave2
...
```

SSH-povezava

Vsi računalniki v gruči morajo imeti SSH-dostop drug do drugega. SSH je okrajšava za secure shell protokol, slovensko protokol varovane lupine [27]. S protokolom SSH se lahko povežemo na oddaljeni računalnik in z njim upravljamo. Preden pa uporabnik dobi pravice za upravljanje nad računalnikom, moramo prestati preverjanje avtentikacije.

Za postavitev okolja za SSH-povezavo sprva generiramo dva RSA-ključa velikosti 2048 bitov, tako dobimo kombinacijo javnega in zasebnega ključa. Oba shranimo v direktorija z imenom "/home/$USER/.ssh". Pika pred imenom direktorija .ssh ponazarja, da je ta neviden. Direktoriju .ssh moramo dodeliti pravice tako, da lahko le lastnik bere, piše in poganja programe v njem. Tako omejimo možnost zlorabe ključev, ki so notri, saj drugi uporabniki računalnika nimajo dostopa do njih. Nato javni ključ razpošljemo na vse računalnike v gruči in prepišemo vsebino javnega ključa v tekstovno datoteko "/home/$USER/.ssh/authorized_keys". Uporabniki, od katerih javne ključe imamo v omenjeni datoteki, imajo nato pravico vzpostavitve SSH-seje z našim računalnikom brez kakršnega koli gesla. Zaradi potrebe računalniške gruče pa morajo imeti računalniki možnost tudi vzpostavitve SSH-seje sami vase.

Posledično vse zasebne ključe, ki jih bo naš računalnik uporabljal, uvozimo v SSH-klient. SSH-agent je program, ki skrbi za vzpostavitev SSH-protokola do drugih računalnikov s strani klienta, na strani strežnika pa moramo imeti program SSH-strežnik, ki sprejme povezavo. Za nastavitve SSH-klienta želimo, da so vedno prisotne, zato jih zapišemo v datoteko "/home/$USER/.bashrc". Vsebina te datoteke se izvrši ob prijavi uporabnika v operacijski sistem, zato vanjo shranimo tudi okoljske (angleško environment spremenljivke) in podobne nastavitve. Na novo zapisane nastavitve v datoteki ".bashrcžačnejo veljati z novo sejo uporabnika. Ročno jih lahko osvežimo z ukazom šource .bashrc". Ali smo vso konfiguracijo pravilno nastavili, lahko preverimo tako, da vzpostavimo SSH-sejo sami s seboj, zato v ukazno lupino vpišemo ukaz "ssh localhost".

Nameščanje Jave in Hadoopa

Omenili smo, da je Hadoop napisan v programskem jeziku Java, zato jo je obvezno tudi namestiti na vsak računalnik, kjer bomo uporabljali Hadoop. Javo najhitreje namestimo s privzetim Linuxovim upravljavcem paketov. Ne smemo pa pozabiti zapisati v okoljsko spremenljivko "JAVA_HOME" pot do njene binarne datoteke. Uspešnost namestitve preverimo tako, da v lupino zapišemo ukaz "java -version", če dobimo informativni odziv o verziji Jave, je ta uspešno nameščena.

Listing 3.3: Namestitev Jave.

```
add-apt-repository ppa:webupd8team/java
apt-get update
apt-get -y install default-jre default-jdk
update-alternatives --config java
```

Ko smo uspešno namestili Javo, lahko iz Apachejevega FTP-strežnika namestimo kompresirano tgz-datoteko, ki vsebuje Hadoop. Datoteko odpremo in njeno vsebino shranimo v "/usr/local/hadoop". Vsebino, ki smo jo dobili s FTP-strežnika, premaknemo v novo ustvarjen direktorij. Ustvariti pa moramo še dva poddirektorija v "/usr/local/hadoop", in sicer "hadoop_tmp/hdfs/namenode"ter "hadoop_tmp/hdfs/datanode". V ta dva pod direktorija bo kasneje Hadoopov daemon kasneje zapisoval začasne podatke, ki jih Hadoop potrebuje za svoje delovanje. Ob koncu namestitve še shranimo Hadoopovo domačo mapo kot okoljsko spremenljivko HADOOP_HOME in dodamo v spremenljivko PATH-pot do Hadoopovih binarnih datotek ter skript. S poslednjim ukazom "hadoop versionše prepričamo, da ni prišlo do nobene napake, če se nam izpišejo informacije o nameščeni distribuciji Hadoopa.

Hadoopove konfiguracijske datoteke

Hadoopove konfiguracijske datoteke so privzeto v Hadoopovem domačem direktoriju "$HOME_HADOOP/etc/hadoop". Vse naslednje datoteke, ki jih moramo urediti za pravilno delovanje HDFS-ja, so v omenjenem direktoriju in so zapisane v XML-formatu.

Odpremo **hadoop-env.sh** in notri prepišemo spremenljivko JAVA_HOME, ki smo jo že nastavili v domači datoteki ".bashrc". Kot že samo ime datoteke nakazuje, lahko tukaj notri nastavimo vse okoljske spremenljivke, ki jih nato Hadoop uporablja pri svojem izvajanju. Prav tako lahko recimo v spremenljivko HADOOP_CONF_DIR prepišemo prevzeto pot do konfiguracijskih datotek.

Naslednja datoteka, ki jo moramo popraviti, je **core-site.xml**. Notri vstavimo podatke o IP-naslovu in vratih, na katerih bo Hadoopov HDFS sprejemal promet. IP lahko zapišemo s spremenljivko "Master", saj bo ta kasneje preslikana v IP-naslov, ki smo ga definirali v datoteki "hosts".

Listing 3.4: Ureditev core-site.xml datoteke.

```
<property>
  <name>fs.default.name</name>
  <value>hdfs://Master:9000</value>
</property>
```

Zadnja konfiguracijska datoteka, ki jo moramo urediti, je **hdfs-site.xml**. Vanjo zapišemo nastavitve, ki se nanašajo na HDFS. Kot prvo nastavitev nastavimo faktor, po katerem nato HDFS replicira podatke po naši gruči. Večji kot je faktor, večja je naša toleranca do napak, vendar moramo v obzir vzeti tudi dejstvo, da se tako veča število redundantnih podatkov. Drugi dve lastnosti določita direktorija, ki ju bosta uporabljala Namenode in Datanode, v njih bo Hadoop shranjeval trajne podatke ter začasne datoteke. Kot zadnjo nastavitev sem izklopil pregled po lastništvu in pravicami nad ripozitoriji ter datotekami, shranjenimi v HDFS-ju.

Listing 3.5: Ureditev hdfs-site.xml datoteke.

```
 <property>
      <name>dfs.replication</name>
      <value>4</value>
</property>
<property>
      <name>dfs.namenode.name.dir</name>
      <value>
          file:/usr/local/hadoop/hadoop_tmp/hdfs/namenode
      </value>
</property>
<property>
      <name>dfs.datanode.data.dir</name>
      <value>
          file:/usr/local/hadoop/hadoop_tmp/hdfs/datanode
      </value>
</property>
<property>
      <name>dfs.permissions</name>
      <value>false</value>
</property>
```

3.1.4 Uporaba Hadoopa

Po ureditvi konfiguracijskih datotek lahko izpišemo vsebino direktorija "/usr/local/hadoop/sbin". Ugotovimo, da so notri datoteke, s katerimi upravljamo s HDFS-gručo. Hadoopov HDFS zaženemo s skripto "start-dfs.sh", ugasnemo ga s "stop-dfs.sh". Preden pa prvič prižgemo HDFS, ga moramo formatirati, to storimo z ukazom "hdfs namenode -format". Ukaz izbriše vse podatke znotraj "/usr/local/hadoop/hadoop_tmp/" in tako HDFS pripravi na uporabo. Če bi ostali kakršnikoli zastareli začasni podatki znotraj omenjenega direktorija iz prejšnjih sej, bi lahko to spravilo sistem v nekonsistentno stanje.

Ob zagonu se na vseh računalnikih začnejo izvajati Hadoopovi Javanski procesi, ki jih lahko izpišemo s pomočjo ukaza "jps".

Ko je sistem HDFS operativen, z njim opravljamo preko računalnika, ki nosi vlogo gospodarja in na katerem teče Namenode. Ukazi za upravljanje s HDFS-jev so preprosti in sintaktično spominjajo na ukaze, ki jih srečamo v Linuxovi ukazni lupini. V spodnji tabeli 3.1 navedimo nekaj primerov osnovnih Hadoopovih ukazov.

primer ukaza	opis
`hadoop fs -mkdir /foo`	V HDFS ustvarimo /foo ripozitorij.
`hadoop fs -ls /`	Izpišemo vsebino. /
`hadoop fs -put /bar.txt /foo`	Prekopiramo iz lokalnega pomnilnika v HDFS-jev /foo.
`hadoop fs -get /foo/bar.txt /test`	Datoteko bar.txt iz HDFS prekopiramo na lokalni /test.
`hadoop fs -rm /foo/bar.txt`	Izbrišemo datoteko bar.txt iz HDFS-ja.

Tabela 3.1: Ilustrativni primeri HDFS ukazov v Linuxovi Bash ukazni lupini.

V spletnem brskalniku odpremo spletno stran na IP-naslovu in vratih 50070, saj na njih Namenode prevzeto servira grafično nadzorno ploščo, na kateri lahko vidimo osnovne informacije o gruči. Vse prikazane informacije lahko dobimo tudi preko ukazne vrstice, vendar nam grafični vmesnik olajša delo.

Na sliki 3.3 lahko vidimo informacije o gruči, ki sem jo postavil za potrebe Comtradove poletne šole EdIT 2016, kjer sem prisostvoval kot inštruktor. HDFS-sistem, ki sem ga postavil, je imel bruto 1.88 terabajtov, od tega neto 1.69 terabajtov. Razliko prinesejo operacijski sistemi in ostali programi, nameščeni po vozliščih gruče.

Configured Capacity:	1.88 TB
DFS Used:	426.04 KB (0%)
Non DFS Used:	193.21 GB
DFS Remaining:	1.69 TB (89.98%)
Block Pool Used:	426.04 KB (0%)
DataNodes usages% (Min/Median/Max/stdDev):	0.00% / 0.00% / 0.00% / 0.00%
Live Nodes	14 (Decommissioned: 0)
Dead Nodes	0 (Decommissioned: 0)
Decommissioning Nodes	0
Total Datanode Volume Failures	0 (0 B)
Number of Under-Replicated Blocks	0
Number of Blocks Pending Deletion	0
Block Deletion Start Time	8. 7. 2016 11:47:13

Slika 3.3: Tabela iz Hadoopovega uporabniškega vmesnika s pregledom HDFS-gruče.

3.2 Spark

Apache Spark je tehnologija, ki se uporablja za procesiranje podatkov na gruči računalnikov [3]. Ponaša se z bliskovito hitrostjo, saj primarno uporablja le RAM-pomnilnik, če imamo preveč podatkov, pa uporabi tudi disk, seveda to zelo negativno vpliva na performans [4]. Odločno se dopolnjuje s HDFS-jem, saj ima že vgrajeno podporo zanj, prav tako pa ga lahko uporabimo v kombinaciji z drugimi podatkovnimi bazami, recimo s Cassandro. Zasnovan je tako, da pokriva kar se da širok spekter nalog, za katerega ga lahko uporabimo, najbolj pogosto pa se pojavlja v kombinaciji z algoritmi umetne inteligence in podatkovnega rudarjenja.

Spark je leta 2009 razvil Matei Zaharia [29], ki trenutno službuje kot asistent na ameriškem inštitutu MIT. Leta 2013 pa je skrb nad projektom prevzela Apachejova fundacija in ga uvrstila med svoje najvidnejše projekte z najvišjo prioriteto. Trenutno Spark v produkciji uporablja približno 1000 podjetij, med njimi izstopajo Amazon, eBay, NASA, Yahoo in drugi. Največja trenutna gruča je zgrajena iz več kot 8000 vozlišč in naj bi procesirala cel petabajt podatkov.

Omenil bi tudi zloglasno tekmovanje, ki je potekalo septembra 2014. Takrat sta se namreč pomerila Hadoop in Spark, kdo od njiju hitreje uredi 100 terabajtov podatkov [31]. Rezultati tekmovanja so bili naslednji:

	Hadoop MapReduce	Spark
Velikost podatkov	102.5 TB	100 TB
Porabljeni čas	72 min	23 min
Število vozlišč	2100	206
Hitrost sortiranja	1.42 TB/min	4.27 TB/s

Tabela 3.2: Rezultati tekmovanja med Hadoopom in Sparkom.

V zgornji tabeli 3.2 je hitro razvidno, da je Spark dominiral in brez težav prehitel gručo Hadoop. Čez palec lahko rečemo, da je Spark le z eno desetino vozlišč nalogo opravil v približno tretjini časa. Apache zatrjuje, da naj bi bil Spark kar 100-krat hitrejši pri operacijah iz pomnilnika in 10-krat pri delu z diskom.

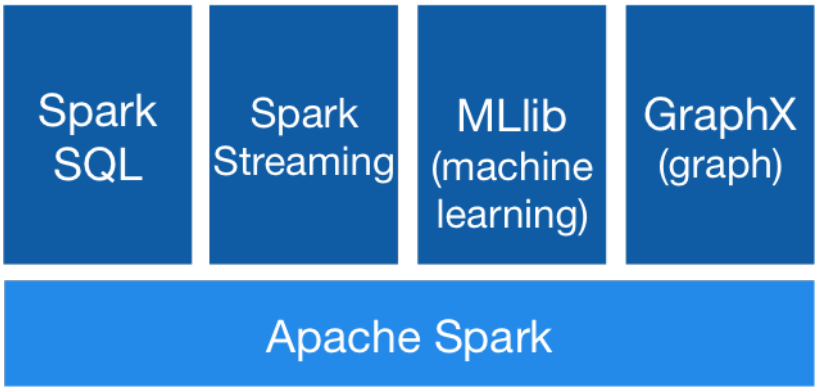
Ob pogledu na zgornjo tabelo pa se nam poraja vprašanje, zakaj bi sploh hoteli uporabljati Hadoop. Odgovor je preprost: Hadoopov HDFS lahko uporabljamo za dolgoročno shranjevanje podatkov, saj Spark sam tega ne omogoča. Spark lahko izkoristimo le za procesiranje podatkov, ki so shranjeni v HDFS-ju. Obe tehnologiji odlično sodelujeta in pametno je, da izkoristimo prednosti obeh.

Spark ima tudi zelo aktivno skupnost, na spletni strani "spark-packages.org" lahko poiščemo dodatne module in implementacije algoritmov, ki jih ni mogoče najti v uradni verziji Sparka.

3.2.1 Arhitektura Sparka

Spark prinaša še nekaj pozitivnih lastnosti, ki jih je vredno omeniti, prihaja skupaj z API-ji za štiri programske jezike, in sicer Java, Scala, Python in R. Za Scalo in Python imamo na voljo še interaktivno programsko lupino, Python programsko lupino lahko z okrajšavo imenujemo PySpark.

Kot smo že povedali, pri Hadoopu vsa moč procesiranja sloni na algoritmu MapReduce. Spark v osnovi vsebuje štiri obsežne module, ki vključujejo praktične funkcije, vsak modul iz svoje domene. Na spodnji 3.4 sliki lahko vidimo shemo, ki prikazuje Sparkovo jedro skupaj s štirimi moduli, ki slonijo nad njim.



Slika 3.4: Sparkova struktura s privzetimi moduli.

Sparkovo jedro in RDD-ji

Kot smo videli na sliki sheme Sparkove strukture 3.4, Sparkovo jedro leži pod ostalimi komponentami in postavlja temelje za vse funkcionalnosti, ki jih Spark premore. Skrbi za paralelno izvajanje operacij na gruči in razdeljuje naloge med vozlišči. Je toleranten do napak in kodo prevaja leno (angleško lazy compile).

Leno prevajanje pomeni, da operacije ne izvrši, dokler to ni absolutno potrebno. Ponuja nam delo s posebno podatkovno strukturo, imenovano RDD-ji, ki jih bom opisal v prihajajočem odstavku, poleg pa omogoča še delo z dvema specifičnima tipoma spremenljivk. Prva se imenuje "broadcast"in druga "accomulator". Vsebina spremenljivke "broadcast"je globalna in njeno vsebino lahko preberemo iz kateregakoli vozlišča gruče, vendar ne moremo vanjo zapisovati, je tipa "read-only". Kontradiktorno lahko iz kateregakoli vozlišča zapisujemo v spremenljivko tipa "accomulator", vendar bere iz nje, lahko pa le tako imenovani program "driven", o katerem bom več napisal kasneje.

Sparkovo jedro je zasnovano na RDD-abstrakciji. RDD je okrajšava za "Resilient Distributed Dataset", ki je osnovna nespremenljiva oziroma (angleško immutable) podatkovna struktura, ki jo Spark uporablja.

S pojmom nespremenljiva ciljamo na lastnost, da ko se enkrat konstruktor objekta izvrši, ga več ne moremo spremeniti. Tako poljubna operacija, ki jo izvršimo nad RDD-jem, ustvari nov RDD ali le prepiše vsebino prejšnjega. Vsak RDD je razdeljen na manjše logične particije, ki so nato porazdeljene po vozliščih gruče, tako nam zagotavlja paralelizacijo operacij.

Nad objekti tipa RDD lahko kličemo dve vrsti funkcij. Prve se imenujejo "transformacije"in druge "akcije". Kadar kličemo funkcijo, ki je tipa transformacija, klicana funkcija le vrne referenco iz starševskega RDD-ja, nad katerim smo klicali omenjeno funkcijo na nov RDD-objekt in operacija, ki bi se mogla izvršiti, se doda v vrsto. Operacija se v tej fazi še ne izvrši, saj smo omenili, da Sparkovo jedro deluje na principu lenega prevajanja kode. Tako lahko naštejemo poljubno število transformacij, a Spark ne bo izvedel zadanih operacij. Ko nad našim RDD-jem kličemo funkcijo tipa akcija, bo RDD izvedel vse prejšnje transformacije, prevedel kodo in nam vrnil rezultat. Naj vse skupaj ponazorimo še s spodnjim ilustrativnim primerom.

Listing 3.6: Primer Sparkovega lenega izvajanja funkcij.

```
array = [1, 2, 3]
rdd = sc.parallelize(array)
rdd = rdd.map(lambda l : l + 1)
rdd = rdd.filter(lambda l : l >= 2 )
result = rdd.collect()
```

Sprva v spremenljivko "arrayžapišemo seznam s tremi števili. V tej fazi je to navadna Pythonova spremenljivka. Nato pa seznam pretvorimo v RDD tako, da jo podamo kot argument funkciji parallelize, ki je funkcija globalnega objekta "sc". Spremenljivka "sc" ponazarja Spark Context [5]. O objektu Spark Context, ki ponazarja enakoimenski program, bomo napisali več kasneje, za sedaj pa se le zavedamo njegovega obstoja. Nato nad vrnjenim RDD-jem kličemo funkcijo "map"in ji kot argument podamo anonimno funkcijo lambda. Map je RDD-jeva funkcija tipa transformacija zato kot rezultat dobimo le referenco na nov objekt, koda pa se v tej fazi še ni izvršila. Nato pa pokličemo funkcijo čollect", ki je tipa akcija. Funkcija collect vrne vsebino RDD-ja v obliki navadnega seznama Python. V tem trenutku, ko Spark prejme funkcijo tipa akcija, izvrši vse prejšnje ukaze in nam v spremenljivko "resultžapiše rezultat. V spodnji tabeli 3.3 tabelo nekaj funkcij, ki sem jih najpogosteje uporabil, seznam in podrobnejši opis vseh funkcij pa je v dokumentaciji Sparka.

Funkcija	Vrsta	Opis
`map(func)`	T	Nad vsakim elementom RDD-ja izvrši podano funkcijo in vrne modificiran element.
`filter(func)`	T	Iterira čez RDD, če funkcija vrne false, izbriše element iz RDD-ja.
`flatMap(func)`	T	Podobno kot map, le da s funkcijo razbije element na več elementov in jih shrani v RDD.
`groupByKey(RDD)`	T	Združi dva RDD-ja glede na enak ključ.
`sortByKey(order)`	T	Uredi RDD-elemente padajoče ali naraščajoče glede na ključ.
`reduce(func)`	A	Agregira elemente tako, da vzame 2 in vrne enega.
`collect(), take(N)`	A	Vračajo elemente, prva vse, druga prvih N.
`count()`	A	Prešteje vse elemente v RDD-ju.
`saveAsTextFile(path)`	A	RDD shrani kot tekstovno datoteko.

Tabela 3.3: Nekaj osnovnih RDD transformacij in akcij. V drugem stolpcu T nakazuje, da je opisujoča funkcija transformacija, in A simbolizira akcijo.

Pri funkcijah “sortByKey” in “groupByKey” v zgornji tabeli 3.3 lahko opazimo delo s ključi. V Sparku v pri kontekstu podatkovne strukture, ki deluje na principu ključev in pripadajočih vrednost, nimamo v mislih tipičnega slovarja. Če v Sparku zapišemo ”tuple”dveh elementov, je prvi element na indeksu 0 ključ, drugi element na indeksu 1 pa predstavlja vrednost.

Sparkovo jedro nam omogoči tudi bolj zahtevno ravnanje z objekti in s pomnilnikom, ki nam je na razpolago. Kot lahko vidimo na spodnjem seznamu, ima Spark različne nivojev shranjevanja, ki jih lahko uporabimo.

1. **MEMORY_ONLY** RDD je shranjen kot neserializiran Javanski objekt v JVM. V primeru prevelikega objekta nekateri deli ne bodo shranjeni v predpomnilniku, bodo pa nemudoma priklicani, če bo potrebno. Ta nivo je privzeti nivo Sparka, saj zagotavlja največjo hitrost procesiranja.

2. **MEMORY_AND_DISK** Podobno kot prvi nivo, le da particije objekta, ki so prevelike, zapišemo na disk.

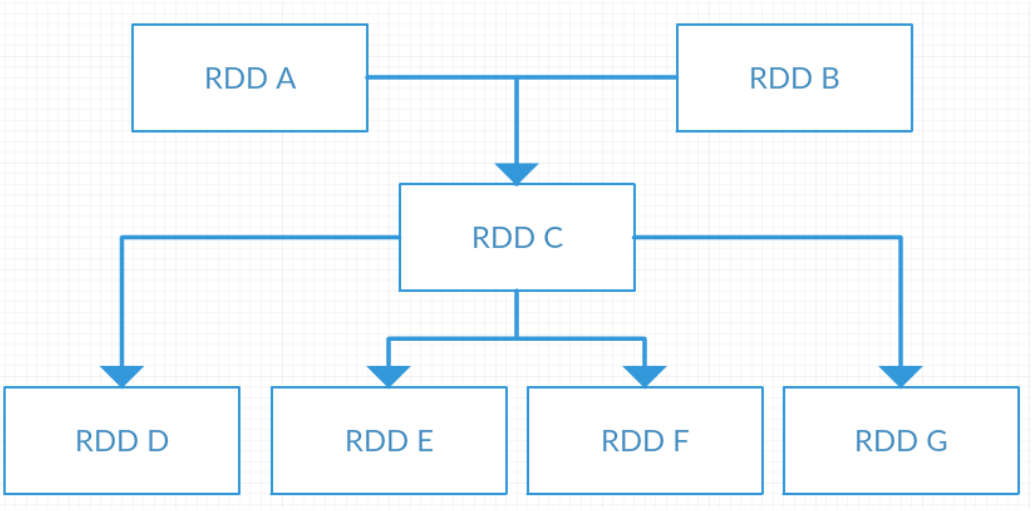
3. **MEMORY_ONLY_SER** Objekt shranimo kot serializiran. Objekt, ki je serializiran, je sicer bolj kompaktno zapisan in ne vzame toliko prostora, vendar je bolj kompleksen za samo procesiranje.

4. **MEMORY_AND_DISK_SER** Podobno kot drugi nivo, le da particije, ki so prevelike, shranimo na disk, tokrat v serilizirani obliki.

5. **DISK_ONLY** RDD shranimo samo na disk.

6. **MEMORY_ONLY_2 in MEMORY_AND_DISK_2** Podobno kot zgoraj opisana nivoja, le da tukaj repliciramo particije objekta na dve različni vozlišči v gruči.

Sparkovi sistemi ponavadi strmijo k temu, da imamo čim več pomnilnika in procesorskih jeder. Da neki objekt shranimo na disk, moramo imeti zelo dober razlog, saj to katastrofalno vpliva na naš performans, tudi če uporabljamo SSD.



Slika 3.5: Shema toka izvajanja programa na Sparku.

Vsak RDD ima tudi funkcijo, imenovano "persist". Ko nad RDD-jem kličemo omenjeno funkcijo, ta RDD shranimo v MEMORY_ONLY nivo pomnilnika. Glede na to, da nimamo neomejenih virov, pa moramo dobro premisliti, kateri RDD je najbolj optimalno shraniti. Ko pogledamo zgornjo sliko 3.5, pa moramo razmisliti, kateri izmed 7 RDD-jev je najbolj primeren za shranitev v predpomnilnik. Če shranimo RDD A, nad katerim smo izvedli samo eno operacijo, bomo celo izgubljali čas s shranjevanjem, zato moramo vedno v predpomnilnik shraniti RDD, nad katerim bomo izvajali več operacij, s čimer optimiziramo izvajanje programa. V našem primeru je to RDD C, saj nad njim izvedemo še vsaj 4 operacije.

SparkSQL

V objekt RDD lahko zapišemo oziroma pretvorimo kakršnokoli podatkovno strukturo, skozi katero je mogoče iterirati. Podatki v RDD-ju so lahko povsem poljubni in se razlikujejo od elementa do elementa. Če imamo ogromne količine podatkov, za katere je bil Spark pravzaprav razvit, se lahko kaj kmalu pojavijo nepričakovane razlike med elementi RDD-ja, zaradi katerih se sesuje naš program. Rešitev problema je strukturiranje podatkov, kar pa nam omogoči modul SparkSQL.

Preden podatke strukturiramo, moramo pripraviti shemo, ki nam služi kot ogrodje. V shemi definiramo podatkovni tip vrednost in pa, ali dopuščamo, da je vrednost lahko nedefinirana ali nična. Nato skupaj združimo podatke in shemo ter kot rezultat dobimo objekt tipa DataFrame. DataFrame bi lahko primerjali s SQL-tabelo, kjer imamo 2-dimenzionalno tabelo vrstic in stolpcev z vrednostnimi. Nad Sparkovo DataFrame tabelo lahko kličemo enostavne funkcije, s katerimi filtriramo vrednost, povečamo vrednost celemu stolpcu ... Lahko rečemo, da je glavna prednost DataFrama pred navadnimi RDD-ji to, da lahko nad njim izvajamo SQL-poizvedbe. Rezultat vsake poizvedbe je navaden RDD. Če v Spark uvozimo tekstovno datoteko, kjer je v vsaki vrstici zapisan JSON, lahko Spark tudi sam razpozna podatkovni tip in ustvari DataFrame brez vnaprej podane sheme.

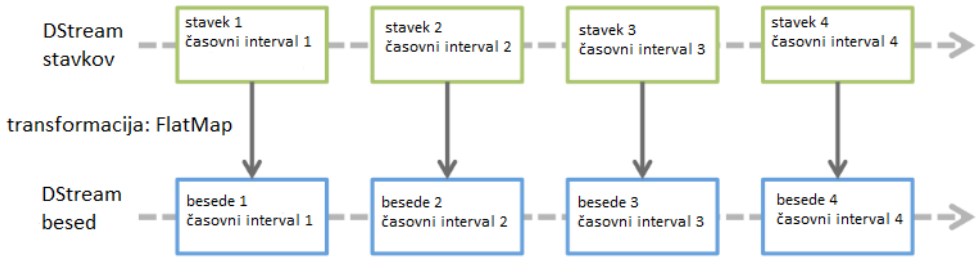
Pri takem avtomatskem definiranju DataFrama moramo biti pozorni, saj včasih Spark napačno definira podatkovne tipe. Do težav pogosto pride, če je JSON gnezden z novim JSON-objektom, tedaj nam notranji JSON prepozna kot navaden string.

Naslednja prednost, ki nam jo prinese SQL-modul, je zapisovanje DataFramov v datoteke s kompresijo Parquet. Če DataFrame tako zapišemo in preberemo nazaj v Spark, kot rezultat dobimo DataFrame in ne RDD, tako se izognemo večkratnemu nepotrebnemu generiranju DataFramov iz podatkov.

Spark Streaming

V praksi pogosto naletimo na izziv, pri katerem moramo konstantno zajemati podatke in le redno konstantno obdelujemo zaključeno množico podatkov. V takih primerih uporabimo Sparkov modul, imenovan Špark Streaming", ki nam omogoča izjemno skalabilno in robustno zajemanje podatkov, posledično obdelavo ter njihovo shranjevanje. Podatke lahko zajemamo iz različnih virov, načeloma pa ločimo dve vrsti virov, in sicer osnovne vire ter napredne vire. Več o njih bom napisal kasneje. Interno pa Spark Streaming deluje tako, da zajete podatke razbije v manjše dele, poimenovane "batchi"in jih pošlje Sparkovemu jedru.

Če na kratko povzamem Spark Streaming, nam ponuja visokonivojsko abstrakcijo našega toka podatkov, poimenovanega DStream. DStream pa si lahko predstavljamo kot poljubno dolgo sekvenco RDD-jev. Nad omenjenimi RDD-ji pa lahko kasneje izvajamo poljubne algoritme in jih tako procesiramo.



Slika 3.6: Izvajanje transformacijske funkcije flatMap na diskretnem toku.

Na začetku poglavja smo omenili, da poznamo napredne in osnovne vire. Pod osnovne vire štejemo datotečni sistem in navadna TCP-vrata oziroma s tujko port. Za zahtevnejše sisteme in aplikacije ponavadi uporabljamo orodja, kot je Apache Kafka, ki nam omogočajo sprejemanje in shranjevanje podatkov iz mnogih poljubnih virov. Tipičen primer uporabe Kafke bi bilo sprejemanje in shranjevanje logov iz več stotih mikroservisov. Apache Spark se lahko poveže s Kafko in procesira prejete podatke. Takemu viru rečemo napreden vir za zajemanje podatkov.

Kot smo videli iz primera štetja na shemi besed 3.6, lahko z DStreamom manipuliramo z več ali manj enakimi funkcijami, kot nam jih ponujajo RDD-ji, prav tako lahko uporabljamo DataFrame in SparkSQL ter MLib.

Seveda pa nam DStream ponuja tudi svojevrstne specifike. Ena izmed njih je poimenovana Windowed Computations in nam omogočajo, da zajamemo RDD-je iz večjega časovnega intervala na DStremu ter jih združimo skupaj kot unijo, tako dobimo manjšo kvantiteto RDD-jev, a vsak izmed njih vsebuje večje število podatkov. Prav tako lahko tokove DStream združujemo ali pa na njih ustvarimo kontrolne točke.

Predstavljajmo si, da nad podatki izvajamo ogromno količino transformacij in na zadnji transformaciji naletimo na nekontrolirano napako, zaradi katere se nam sesuje program, ki se izvaja na gruči Spark. Posledici tega bi bili izjemna potrata virov in izguba podatkov. Zato lahko z enostavnim konceptom kontrolnih točk podatke v neki fazi v toku izvajanja zapišemo na HDFS ter jih tako shranimo.

Sparkovo MLib

Na začetku MLibove [18] dokumentacije je zapisano, da je cilj modula MLib algoritme strojnega učenja narediti enostavne in kar se da skalabilne. Sam modul MLib je razdeljen na dva dela:

1. spark.mllib: Starejši osnovni del modula, zgrajen na RDD-podatkovni strukturi.

2. spark.ml: Novejši in višji del modula, ki ga uporabljamo v kombinacijami z DataFrami iz modula SparkSQL. Zgrajen je nad originalnim Spark.mlib-om.

Spark.mlib je bil do Spark verzije 2.0 primarni del MLiba, z novejšimi verzijami pa se je Apache osredotočil na razvoj drugega dela modula Spark.ml. Spark.mlib bo v prihodnje še vedno vzdrževan, vendar se vanj naj ne bi več implementiralo novih funkcionalnosti.

Spark.mlib

V Spark.mlib, ki stoji nad RDD-ji, lahko najdemo vse pogostejše funkcije, ki jih uporabljamo pri umetni inteligenci oziroma strojnemu učenju. Ponuja nam tudi metode za ocenjevanje modelov in razdelitev učne oziroma testne množice. V spodnji ilustrativni kodi lahko vidimo, da je sama raba modula spark.mlib precej podobna ostalim knjižnicam s podobnim namenom.

Listing 3.7: Aplikativna uporaba odločitvenega drevesa.

```
from pyspark.mllib.tree import DecisionTree
from pyspark.mllib.util import MLUtils

data = MLUtils.loadLibSVMFile(SparkContext, './data.txt')
(X, y) = data.randomSplit([0.7, 0.3])

model = DecisionTree.trainClassifier(X, numClasses=2)

predictions = model.predict(y.map(lambda i: i.features))
```

Glede na to, da imamo ob rabi Sparka vedno v mislih tudi samo skalabilnost sistema, ima Spark.mlib tudi podporo redkih vektorjev in matrik. Redke oziroma "sparse" vektorje in matrike potrebujemo tedaj, ko imamo podatke z veliko enakimi ali ničnimi vrednostnimi. Če si zapomnimo le vrednosti, v katerih imamo zapisano informacijo, lahko izrazito optimiziramo shranjevanje in privarčujemo na pomnilniku.

Spark.ml

Kot sem že omenil, je Spark.ml zgrajen nad Spark.mlib-om, dotika pa se tudi SparkSQL-a, saj nam omogoča izvajanje algoritmov strojnega učenja nad DataFrami. Zaradi svojevrstnih karakteristik Spark.ml-ja moramo omeniti še nekaj novih pojmov.

- **Transformer:** funkcija, ki transformira DataFrame v nov DataFrame. Primer transformerja je model strojnega učenja, ki transformira učne množice DataFrame v nov DataFrame, ki vsebuje informacije o predikcijah. Tehnično pa mora objekt tipa transform implementirati funkcijo transform, preko katere nato dosežemo transformacijo.

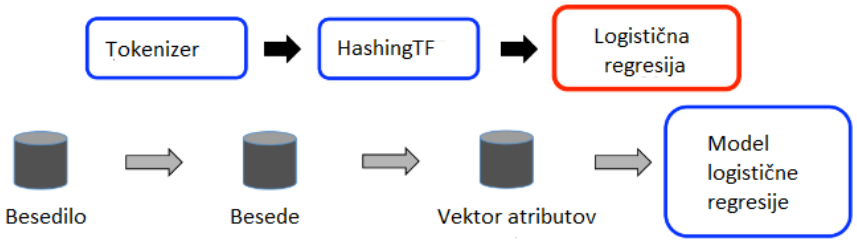
- **Estimator:** Funkcija, ki jo pokličemo na DataFramu in tako proizvedemo Transformer. Za primer: algoritem strojnega učenja, ki iz DataFrame proizvede model za napovedi, je estimator. Estimator je primoran implementirati metodo fit.

- **Pipeline:** Pri strojnem učenju ponavadi vežemo sekvenco funkcij ter tako ustvarimo proces, pri katerem transformiramo podatke. Zato uporabimo pipeline, saj z njim vežemo večje število transformerjev in estimatorjev skupaj ter tako definiramo daljši celoviti tok izvajanja.

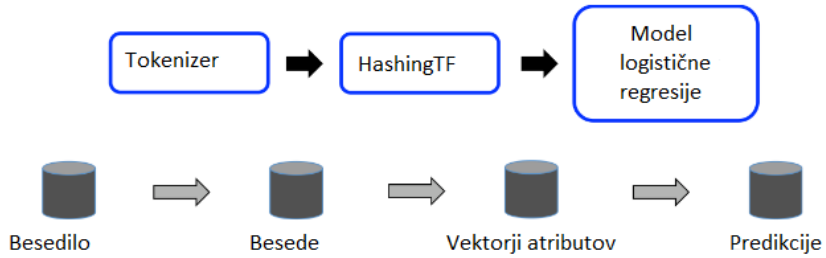
- **Parameter:** Vsi transformatorji in estimatorji imajo poseben unikaten atribut, poimenovan ID, in si delijo skupni API za definiranje parametrov.

Po navadi sprva definiramo pipeline ter tako dobimo vrsto, za katero lahko definiramo prihodnjo sekvenco dogodkov oziroma korakov. Sekvenca je vedno izvedena v enakem zaporedju in na vsakem koraku imamo transformer ali estimator. Za transformer izvedemo funkcijo transform(), za estimator pa fit(). Transformer nam vrne nov DataFrame, estimator pa Transformer. Vsak transformer, ki ga vrne estimator, postane del tega istega pipelina.

Pipeline se lahko konča s transformatorjem ali pa z estimatorjem, kar pomeni, da bo rezultat celega izvajanja transformator ali pa nov DataFrame. Glede na to delimo pipeline v dve kategoriji, in sicer prvo kategorijo imenujemo Pipeline, če se konča z estimatorjem, ter drugo kategorijo PipelineModel, če se konča s transformatorjem. Če je zadnji korak pipelina estimacija, dobimo transform oziroma model za predikcijo, obratno če je zadnji korak transformacija, pa je poslednji rezultat sama predikcija.



Slika 3.7: Primer navadnega Pipelina, katerega rezultat je esitmator.



Slika 3.8: Primer PipelineModela, saj je zadnji korak Pipelina DataFrame s predikcijami.

GraphX

Zadnji Sparkov modul se imenuje GraphX. Iz imena je razvidno, da je modul namenjen delu z grafi. Kot vsi predhodni moduli je tudi ta izpeljan iz Sparkove osnovne podatkovne strukture RDD.

Z modulom GraphX na RDD-je vpeljemo na kar se da praktičen in enostaven način abstrakcijo grafov. Grafi tako podedujejo lastnosti RDD-jev in so nespremenljivi, distribuirani ter odporni proti napakam. GraphX je v nekaterih aspektih bolj optimiziran od RDD-jev; na primer: če spremenimo eno vozlišče, se to prepiše v nov graf, ostalim nedotaknjenim vozliščem pa se le prenese referenca. Za razliko od navadnih RDD-jev, pri katerih smo primorani prepisati prav vsa polja oziroma podatke.

GraphX ponuja metode za lažje ustvarjanje grafov, seveda pa so nam na voljo tudi osnovne operacije, med njimi recimo iskanje podgrafov, združevanje vozlišč in podobno. Za tiste bolj zahtevne uporabnike pa je na voljo tudi sam optimiziran API Preglovega sistema za procesiranje večjih grafov. Vsako vozlišče grafa ima 64-bitno identifikacijsko številko, poimenovano VertexID, to pa ob enem omejuje tudi maksimalno velikost grafa na natanko 2^{64} vozlišč.

Najbolj pogosta praktična uporaba modula GraphX je s tako imenovanimi "Property graph"oziroma lastnostnimi grafi, kjer vsakemu vozlišču dodamo poljuben objekt tako, da simbolizira oziroma nosi informacijo o neki lastnosti.

Podatke o vozliščih in relacijah med njimi hranimo v ločenih tabelah, skupaj pa zna to GraphX združiti in interpretirati kot graf.

Na žalost pa ima GraphX podpira samo Scala API, zato ga ne moremo uporabljati v kombinacijami s Pythonom. Odlična alternativa je Sparkov paket, imenovan GraphFrames, ki ponuja API-je za Javo, Scalo in Python.

3.2.2 Konfiguracija Sparka

Konfiguracijo omrežja ter protokola SSH za povezovanje med računalniki smo naredili že pri nameščanju Hadoopa 3.1, zato lahko ta del tukaj preskočimo.

Namestitev Scale in Sparka

Spark je razvit v programskem jeziku Scala, zato je namestitev tega obvezna za uporabo Sparka. Scale ni mogoče namestiti s privzetim Linuxovim Advanced Packaging Tool oziroma krajše APT-jem. APT-je Linuxov privzeti program za upravljanje s paketi oziroma programi, ki jih želimo namestiti s seznama naših virov oziroma naslovov strežnikov shranjenih v datoteki /etc/apt/sources.list. Scalo moramo zato ročno namestiti s Scalovih FTP-strežnikov. Kot sem zapisal že pri namestitvi Hadoopa, moramo imenik, kjer so binarne datoteke, dodati v PATH-Linuxovo spremenljivko. Uspešno namestitev preverimo z ukazom ščala -version", ki nam izpiše naloženo verzijo Scale, če je bila namestitev uspešna.

Listing 3.8: Postopek namestitve Scale.

```
mkdir -p /usr/local/scala ; chown -R ubuntu /usr/local/scala
wget http://www.scala-lang.org/files/archive/scala-2.9.0.final.tgz
tar -xzvf /home/ubuntu/scala-2.9.0.final.tgz
cp -r /home/ubuntu/scala-2.9.0.final/* /usr/local/scala
echo export SCALA_HOME=/usr/local/scala >> /home/ubuntu/.bashrc
echo PATH=$PATH:$SCALA_HOME/bin >> /home/ubuntu/.bashrc
/bin/bash -c "source /home/ubuntu/.bashrc"
```

Podobno kot Scalo namestimo še Spark z Apachijevih FTP-strežnikov. Pomembno je, da naredimo tudi delovni direktorij ter direktorij za shranjevanje zapiskov oziroma logov. Dobra praksa pa je, da zapiske Spraka shranjujemo na HDFS ali na kakšen drug strežnik, v nasprotnem primeru vsak računalnik v gruči shranjuje svoje zapiske na svoj lokalni pomnilniški sistem, to pa močno oteži njihovo prebiranje, saj moramo hkrati prebirati več datotek zaradi paralelnega izvajanja programa po gruči.

Konfiguracija Sparka

Sparkove konfiguracijske datoteke lahko najdemo v Sparkovem domačem v direktoriju $SPARK_HOME/conf. Za postavitev gruče sem moral urediti tri glavne konfiguracijske datoteke.

Prva datoteka se imenuje **slaves**. S povsem enako datoteko smo se srečali že pri konfiguraciji Hadoopa. Vanjo zapišemo naslove računalnikov, ki nam bodo služili kot sužnji. Naslove zapišemo z IP-ji ali vzdevki iz hosts datoteke tako, da lahko nato gospodar s protokolom SSH z njimi vzpostavi povezavo in na njih izvaja naloge.

Druga konfiguracijska datoteka je **spark-defauls.conf**, kamor zapišemo najbolj pomembne nastavitve. Tako določimo gospodarjevo vozlišče, direktorij za zapiske ter dodamo kakšne tretje knjižnice. Nastavitve pišemo kot vnaprej definirane ključe in pripadajoče vrednosti.

Z nastavitvijo ključa “spark.eventLog.dir” Sparku definiramo lokacijo, kamor naj shranjuje loge. Trivialen primer bi bil /usr/local/spark/logs, če pa namesto lokalnega direktorija zapišemo neki zunanji vir, v našem primeru URI od ”hdfs://Master:9000/spark_logs”, bo Spark prepoznal, da referenciramo HDFS, natančneje imenik /spark_logs ter loge.

Prav tako vključimo zunanje dodatke ali knjižnice, ki jih potrebujemo za uspešno izvajanje. To storimo tako, da pod “spark.executor.extraClassPath” in “spark.driver.extraClassPath” dadamo pot do jar-datoteke.

Tretja datoteka je imenovana **spark-env.sh**. V njej so okoljske spremenljivke, iz katerih Spark vzame vrednosti, ki jih nato uporabi pri svojem izvajanju. Seveda bi lahko okoljske spremenljivke naložili v operacijski sistem tudi iz kakšne druge datoteke, vendar Spark pred zagonom samodejno prebere vrednosti iz te datoteke.

Tako tukaj nastavimo SCALA_HOME, ki nam pove direktorij do binarnih datotek Scale. Definirati moramo tudi mrežne nastavitve. Tako v SPARK_MASTER_IP in SPARK_MASTER_PORT zapišemo IP-naslov ter vrata, na katerih Master posluša.

Prav tako je pametno, da omejimo vire, ki jih Spark porablja pri svojem izvajanju. Tako lahko v spremenljivki SPARK_WORKER_CORES omejimo število procesnih niti ter v SPARK_WORKER_MEMORY določimo maksimalno število pomnilnika, ki ga delavski proces lahko uporabi.

Na vsaki napravi lahko poganjamo več procesov Spark, število določimo v spremenljivki SPARK_WORKER_INSTANCES.

Pri postavitvi gruče za poletno šole EdIT 2016 sem na vsakem računalniku imel en proces, ki je uporabljal 3 niti in 3 GB pomnilnika. Kot poslednja, obvezno nastavitev shranimo v SPARK_WORKER_DIR, ki Sparku pove, kateri je njegov delavni direktorij. Obvezno je, da ima Spark vse pravice in nadzor nad direktorijem, da lahko vanj bere in piše ter izvaja programe.

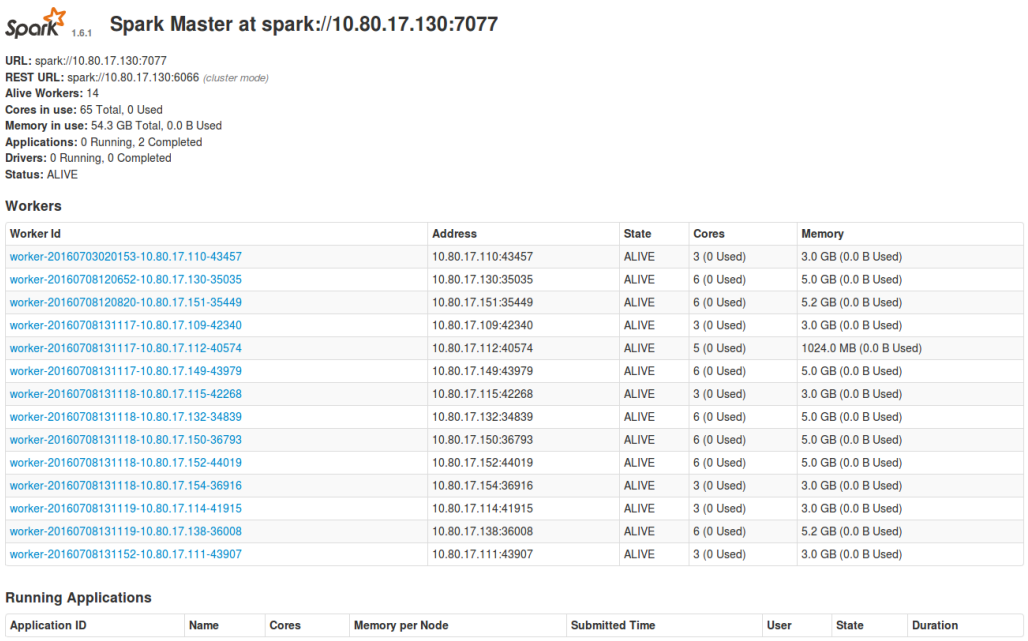
3.2.3 Uporaba Sparka

Po uspešni konfiguraciji lahko celotno gručo zaženemo z dvema skriptama. Sprva moramo zagnati vozlišče z gospodarjem ter nato vozlišča s sužnji, da se lahko ta uspešno povežejo na gospodarja.

Gospodarja zaženemo s skripto $SPARK_HOME/sbin/start-master.sh, vse sužnje pa z $SPARK_HOME/sbin/start-slaves.sh. Ugašanje poteka enako, le da uporabimo skripte s prepono stop v obratnem vrstnem redu, sprva ustavimo sužnje ter nato gospodarja.

Ko je Spark aktiven, lahko nanj preko Spark Contexta pošiljamo programe, ki se nato izvedejo na vseh vozliščih. Program pošljemo na gospodarjevo vozlišče in izbrana vrata, prevzeta vrata so 7077. Na gručo se lahko povežemo tudi z interaktivno Scala ali Python ukazno lupino. Odvisno od naših nastavitev Spark procesira zadane naloge paralelno ali pa eno naenkrat. Vsak program se začne izvajati nemudoma, ko ima gruča na voljo dovolj zahtevanih virov.

Za bolj transparentno delovanje nam Spark na gospodarjevem vozlišču na privzetih vratih 8080 servira nadzorno ploščo, kjer lahko vidimo informacije o trenutnem stanju gruče. Prav tako vsako suženjsko vozlišče servira svojo nadzorno ploščo na vratih 8081. Pregled imamo nad trenutnimi viri, izvajajočimi se procesi in pa procesi, ki so se že izvedli. Vidimo, koliko virov naša gruča kje uporablja, še več, lahko pogledamo tudi vsak proces po sebi ter kako je Spark transformiral RDD-je.



Spark 1.6.1 **Spark Master at spark://10.80.17.130:7077**

URL: spark://10.80.17.130:7077
REST URL: spark://10.80.17.130:6066 *(cluster mode)*
Alive Workers: 14
Cores in use: 65 Total, 0 Used
Memory in use: 54.3 GB Total, 0.0 B Used
Applications: 0 Running, 2 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers

Worker Id	Address	State	Cores	Memory
worker-20160703020153-10.80.17.110-43457	10.80.17.110:43457	ALIVE	3 (0 Used)	3.0 GB (0.0 B Used)
worker-20160708120652-10.80.17.130-35035	10.80.17.130:35035	ALIVE	6 (0 Used)	5.0 GB (0.0 B Used)
worker-20160708120820-10.80.17.151-35449	10.80.17.151:35449	ALIVE	6 (0 Used)	5.2 GB (0.0 B Used)
worker-20160708131117-10.80.17.109-42340	10.80.17.109:42340	ALIVE	3 (0 Used)	3.0 GB (0.0 B Used)
worker-20160708131117-10.80.17.112-40574	10.80.17.112:40574	ALIVE	5 (0 Used)	1024.0 MB (0.0 B Used)
worker-20160708131117-10.80.17.149-43979	10.80.17.149:43979	ALIVE	6 (0 Used)	5.0 GB (0.0 B Used)
worker-20160708131118-10.80.17.115-42268	10.80.17.115:42268	ALIVE	3 (0 Used)	3.0 GB (0.0 B Used)
worker-20160708131118-10.80.17.132-34839	10.80.17.132:34839	ALIVE	6 (0 Used)	5.0 GB (0.0 B Used)
worker-20160708131118-10.80.17.150-36793	10.80.17.150:36793	ALIVE	6 (0 Used)	5.0 GB (0.0 B Used)
worker-20160708131118-10.80.17.152-44019	10.80.17.152:44019	ALIVE	6 (0 Used)	5.0 GB (0.0 B Used)
worker-20160708131118-10.80.17.154-36916	10.80.17.154:36916	ALIVE	3 (0 Used)	3.0 GB (0.0 B Used)
worker-20160708131119-10.80.17.114-41915	10.80.17.114:41915	ALIVE	3 (0 Used)	3.0 GB (0.0 B Used)
worker-20160708131119-10.80.17.138-36008	10.80.17.138:36008	ALIVE	6 (0 Used)	5.2 GB (0.0 B Used)
worker-20160708131152-10.80.17.111-43907	10.80.17.111:43907	ALIVE	3 (0 Used)	3.0 GB (0.0 B Used)

Running Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration

Slika 3.9: Sparkova nadzorna plošča gruče, ki sem jo postavil na poletni šoli EdIT 2016.

Poglavje 4

Priprava in vizualizacija podatkov

V uvodu poglavja 2 smo opisali tehnologije, ki smo jih uporabil pri zbiranju podatkov, in vire. V nadaljevanju sem opisal obe tehnologiji, ki sem ju uporabil za postavitev računalniške gruče. Ko sem imel postavljeno gručo in zbrane podatke, sem jih začel obdelovati. Za obdelavo podatkov sem uporabljal skriptni jezik Python2.7.

Listing 4.1: Primer odgovora iz JCDecaux API-ja za eno postajo BicikeLj.

```
{
        "number": 4,
        "name": "CANKARJEVA UL.-NAMA",
        "address": "Cankarjeva cesta 1",
        "position": {
                "lat": 46.052431,
                "lng": 14.503257
        },
        "banking": false,
        "bonus": false,
        "status": "OPEN",
        "contract_name": "Ljubljana",
        "bike_stands": 26,
        "available_bike_stands": 20,
        "available_bikes": 6,
        "last_update": 1471098509000
}
```

4.1 Migracija podatkov

Podatke smo prvotno shranjevali v podatkovno bazo MongoDB, zato smo jih bili primorani prepisati v HDFS, da so bili tako dostopni Sparku. Eksperimentiral sem z nekaj neuradnimi gonilniki, ki bi povezali Spark neposredno na MongoDB, vendar na začetku pomladi 2016 še ni bilo uradnega orodja, ki bi to zanesljivo omogočalo. Uradni konektor MonoDB-Spark je bil javno objavljen 28. 6. 2016, vendar takrat sem že zaključil s procesom migracije podatkov.

4.1.1 Uvoz podatkov v HDFS

Za branje podatkov iz MongoDB smo uporabili skripto napisano v Pythonu in knjižnico Pymongo. Podatke smo prepisali v tekstovno datoteko, tako da je bil v vsaki vrstici en JSON, ki je vseboval podatke enega API-klica. Podatke smo že v tem koraku prvič delno prefiltrirali in odstranili vse očitno nepotrebne atribute. Primer nepotrebnega atributa bi bila unikatno identifikacijska številka, ki jo MongoDB doda vsem elementom v bazi, prav tako NPM-modul Mongoose doda verzijo ključa, ki ga uporablja, prisotni pa so bili tudi ostali metapodatki, recimo interne oznake oskrbovalcev API-jev. Pretvorili smo tudi Unixovo časovno oznako Epoch, iz milisekund v sekunde, tako smo se znebili nepotrebne natančnosti in privarčevali pri pomnilniku.

Listing 4.2: Vspostavitev povezave z MongoDB in prebiranje vseh dokumentov iz kolekcije BicikeLj v Pythonov seznam.

```
from pymongo import MongoClient

client = MongoClient('mongodb://user:pass@10.10.4.5:27019/apidata')
database = client["apidata"]
bicikeljArray = database["bicikeljs"].find({})
```

Nato je bilo treba tekstovne datoteke s podatki kopirati na vozlišče gospodarja HDFS-gruče. Kopiranje sem opravil s SCP. SCP je okrajšava za orodje Secure Copy Protocol, omogoča nam varen prenos datotek preko mreže. Po kopiranju sem na HDFS-ju ustvaril direktorij za podatke in jih prekopiral z ukazom:

Listing 4.3: Kreiranje HDFS direktorija in kopiranje podatkov iz lokalnega pomnilnika na HDFS.

```
scp ./bicikelj.txt ubuntu@master:/home/ubuntud

hadoop fs -mkdir /data
hadoop fs -copyFromLocal /home/ubuntu/bicikelj.txt /data
```

4.1.2 Pretvorba podatkov v DataFrame

Ko podatke preberemo iz HDFS v Spark, se ti zapišejo tako, da je vsaka vrstica tekstovne datoteke en element tipa string v RDD-ju. V poglavju, kjer smo opisovali modul SparkSQL, smo zapisali, da je delo z DataFrami enostavnejše in hitrejše kot z osnovnimi RDD-ji, zato smo podatke pred obdelavo pretvorili v RDD-je.

Podatki, prebrani iz JCDecaux API-ja, so bili dobro strukturirani, so pa vsebovali nekaj informacij, ki jih pri prihodnjem delu ne bi mogel uporabiti. Odstranil sem podatke o imenu pogodbenega naročnika sistema, hišni številki postaje ... Končni cilj diplomske naloge je napovedovanje polnosti postaje, zato sem združil podatke o kolesih na postaji in jih delil s številom vseh mest za vrnitev kolesa. Novo dobljeni atribut sem poimenoval "availibility"oziroma razpoložljivost, ki nam v odstotkih pove polnost postaje. V DataFrame sem tako zapisal čas API-klica, identifikacijsko številko postaje in razpoložljivost, predstavljeno v odstotkih.

Listing 4.4: Pretvorba RDD-ja v DataFrame z definirano shemo.

```
from SparkConnector import sparkContext, hadoopMaster
spark_context, sql_context = sparkContext()

bLj_RDD = spark\_context.textFile('{}/bicikelj.txt'.format(hadoopMaster))

bLj_RDD = bicikeljRDD.map(transform).flatMap(transformSplit)

from pyspark.sql.types import StructType, StructField, \
                              FloatType, ByteType, IntegerType

bicikeljSchema = StructType([
    StructField("requestedtime", IntegerType(), False),
    StructField("id", ByteType(), False),
    StructField("availibility", FloatType(), False),
])

bicikeljDF = sql\_context.createDataFrame(bLj\_RDD, bicikeljSchema)
```

Z vsakim klicem na API od OWM sem dobil podatke iz 40 vremenskih postaj, ki so v Ljubljani. Lokacije vsake vremenske postaje ni mogoče eksaktno določiti, saj OWM pridobiva podatke o lokalnem vremenu tudi iz privatnih amaterskih vremenskih postaj. Zaradi enakega razloga lahko v vremenskih podatkih pričakujemo tudi veliko šuma. Podatki so med drugim vsebovali kar nekaj opisnih atributov, ki so se lahko od postaje do postaje razlikovali, saj je kakšna postaja opisovala trenutno stanje kot "oblačno", druga pa kot "sončno".

Vse opisne atribute sem ignoriral. Osredotočil sem se samo na atribute, za katere lahko sklepamo, da so bili najbolj konsistentni in objektivni na vseh postajah. Tako sem za vsak API-klic iz vsake postaje vzel temperaturo zraka, vlago v zraku, zračni pritisk in hitrost vetra ter iz njih izračunal povprečje za vso Ljubljano. V DataFrame sem tako zapisal informacije o času in izračunana štiri povprečja.

Rezultat sta bili dve ločeni tabeli s podatki, ki jih je bilo treba združiti glede na čas. Informacije o polnosti postaj BicikeLj sem zajemal vsako minuto, vremenske podatke pa na tri ure, zato je tukaj prihajalo do neskladja. Tabeli smo združili tako, da vsakemu klicu o polnosti postaje pripišemo časovno najmanj oddaljeno stanje vremena, ki ga poznamo.

Vse ustvarjene DataFrame nato shranimo na HDFS v formatu parquet, tako so pripravljeni za prihodnjo rabo.

4.2 Statistika podatkov

Podatke, ki so zapisani v formatu DataFrame, lahko hitro analiziramo z vgrajenimi funkcijami, ki kot rezultat vrnejo tabelo z osnovnimi statističnimi informacijami o podatkih.

	razpoloženje	temperatura zraka	vlažnost zraka	zračni pritisk	hitrost vetra
# vzorcev	3967261	610	610	610	610
povprečje	0.29	13.96	81.13	946.83	1.22
minimum	0	1.78	65.45	936.24	0.95
maksimum	1	23.30	97.62	960.50	1.97
st. odklon	0.25	4.77	7.86	4.38	0.18

Tabela 4.1: Statističnih podatkov o zajetih atributih.

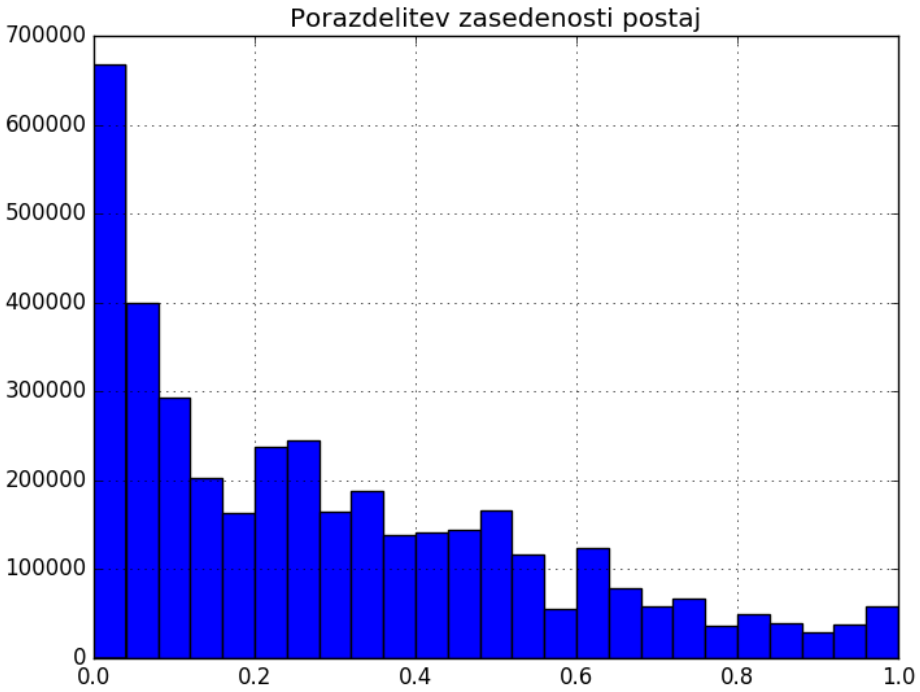
V spodnji tabeli 4.1 vidimo korelacijo med zasedenostjo postaje in našimi atributi. Dodanih je še nekaj drugih atributov, izvzetih iz Linuxove časovne oznake Epoch. Korelacije so zelo nizke oziroma jih skoraj ni. Razumljivih bi bila lahko le 0.038 korelacije med dnevom v tednu in polnostjo postaje. Bolj pozno v tednu kot smo, bolj so postaje polne, saj ljudje čez vikend ne hodijo v službe, šole ...

atribut	korelacija
# temperatura	0.06
pritisk	0.04
dan tedna	0.038
minuta ure	0
vlažnost zraka	0
hitrost vetra	-0.02
ura dneva	-0.02

Tabela 4.2: Korelacije med zasedenostjo postaje in ostalimi atributi.

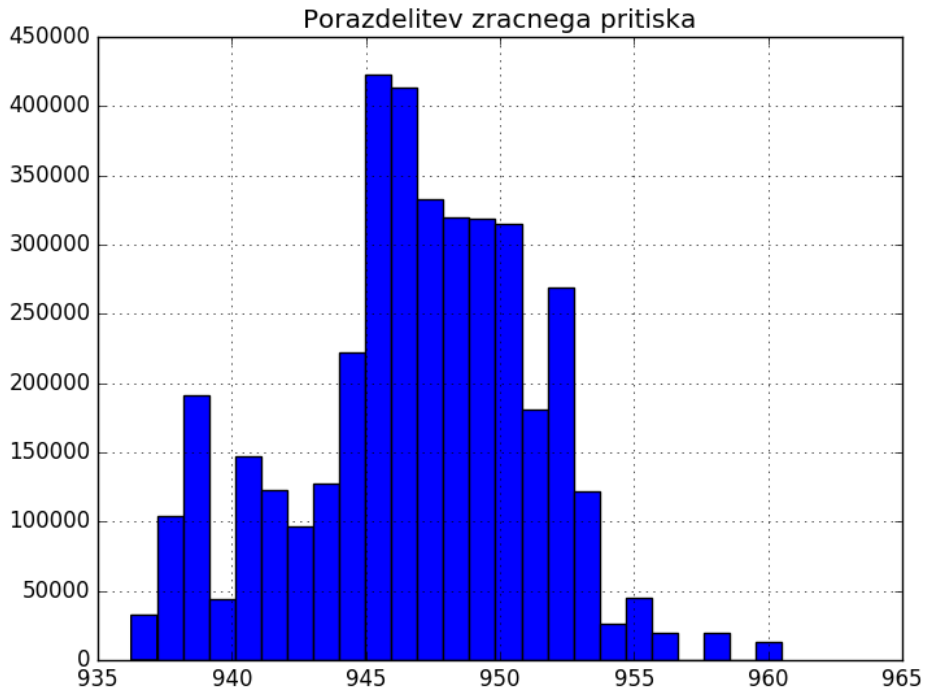
4.3 Porazdelitve

V tabeli, kjer opisujemo statistične podatke o atributih 4.1, opazimo, da je povprečna zasedenost postaje BicikeLj zgolj 29-odstotna, kar na prvi pogled deluje malo. Povedati moramo, da JCDecaux in ostala podjetja gradijo podobne sisteme za izposojo prevoznih sredstev tako, da je dvakrat več parkirnih mest za kolesa kot koles samih. V sistemih, kjer stranka plačuje za najem kolesa ali avtomobila glede na izposojeni čas, je veliko bolj konfliktna situacija, če stranka izposojenega sredstva ne more vrniti, kot pa če si ga ne more izposoditi.



Slika 4.1: Porazdelitev zasedenosti postaj.

Po pričakovanjih porazdelitev ostalih vremenskih atributov bolj ali manj spominja na Gaussovo.

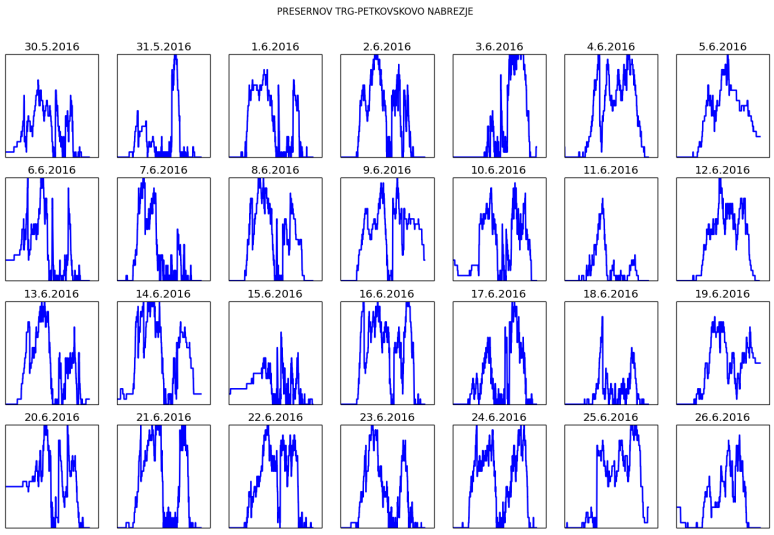


Slika 4.2: Porazdelitev zračnega pritiska.

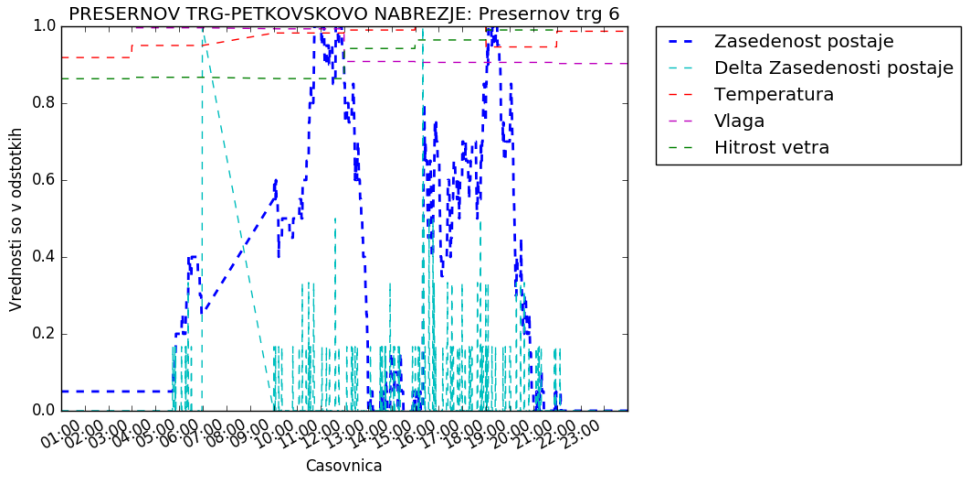
4.4 Sezonskost postaj

Logično je pričakovati, da se bodo razni vzorci obnašanja postaj BicikeLj izrazili skozi tedenske intervale. Torki bodo najbolj podobni torkom ter si delili sorodne lastnosti z ostalimi delavniki. Vikendi se razlikujejo od delavnikov. Prazniki so podobni nedeljam. Zato sem za vsako postajo pripravil vizualizacijo, kjer sem prikazal njene zaporedne tedne. Na sliki 4.3 vidimo štiri tedne najbolj popularne postaje v sistemu BicikeLj, to je postaja na Prešernovem trgu. Graf je izrisan tako, da so ponedeljki v prvem stolpcu, torki v drugem in tako vse do nedelje v sedmem stolpcu.

Ob prvem pogledu na spodnjo vizualizacijo 4.3 opazimo vzorce. Postaja je ponoči večinoma prazna in se dopoldan napolni. V popoldnevu lahko pričakujemo praznitev, cikel polnitve in praznitve se še enkrat ponovi. Vidna je tudi razlika med sobotami in nedeljami ter delavniki. Izrazito podobni so si četrtki, ko lahko sredi dneva pričakujemo hitro in popolno izpraznitev ter napolnitev postaje.

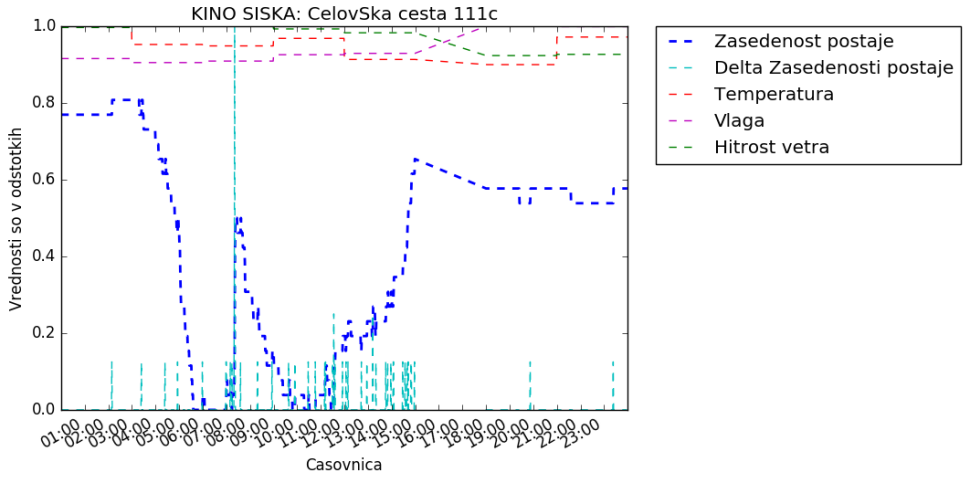


Slika 4.3: Zasedenosti postaje na Prešernovem trgu skozi štiri zaporedne tedne.



Slika 4.4: Vizualizacija srede, 25. 5. 2016, na Prešernovi postaji.

Sumljive so nenadne velike praznitve ali polnitve. Lahko jih pripišemo delavcem, ki razvažajo kolesa, vendar ne moremo biti prepričani, saj nimamo podatkov o njihovem delu. Na zgornji sliki 4.4 praznitve ob približno 14:00 ne pripisujem kombiju, saj se je postaja izpraznila v približno petih minutah. Na drugi strani pogledamo vizualizacijo 4.5 in lahko zagotovo sklepamo, da so delavci napolnili postajo Kino Šiška v ponedeljek, 23. 5. 2016, ob približno 8:00, saj se je v eni minuti postaja napolnila z 0 na natanko 50 odstotkov.



Slika 4.5: Vizualizacija ponedeljka, 23. 5. 2016, na postaji Kino Šiška.

4.5 Gradient zasedenosti postaje

Na zgornjih vizualizacijah postaje na Prešernovem trgu in postaje Kino Šiška smo opazili, da se lahko postaje izjemno rapidno izpraznijo ali napolnijo.

Ko izračunamo minimalni, maksimalni, povprečni in standardni odklon minutnega ter petminutnega gradienta, dobimo naslednje povprečne vrednosti.

povprečja	minutni gradient	petminutni gradient
maksimum	0.5096	0.6313
povprečje	0	0
minimum	-0.626	-0.6583
standardni odklon	0.0172	0.0578

Tabela 4.3: Povprečja gradientov.

Kot razberemo iz tabele 4.3, se lahko v največjih ekstremih postaja v eni minuti napolni za dobrih 50 odstotkov ali izprazni za dobrih 62 odstotkov.

Izstopali sta postaji na Trgu osvobodilne fronte na Kolodvorski ulici ter postaja na Trgu delavnih brigad, saj sta bili edini dve postaji, ki sta se v manj kot petih minutah popolnoma izpraznili.

4.6 Združevanje podobnih postaj

Intuitivno je, da se na postajah, ki so geografsko locirane skupaj ali pa si delijo neko lastnost, to odraža tudi na njihovem ritmu izposoj. Če podrobno pogledamo na dnevni vizualizacijo od postaje na Prešernovem trgu 4.3 in vizualizacijo postaje pri Kinu Šiška 4.5, opazimo, da sta si povsem komplementarni.

Postaja Kino Šiška je locirana izven strogega centra, v večernih urah se napolni in v jutranjih urah izprazni. Ljudje se s postaj, ki so okoli centra in so tipično postavljene okoli gosto naseljenih sosesk, v jutranjih urah odpeljejo v center ter kolo pustijo na postaji v centru, kot je Prešernov trg. Zvečer si za pot nazaj domov izposodijo kolo v centru in ga vrnejo na postaji, ki je najbližja njihovemu domu. Tako bi lahko opisali dnevni ritem sistema BicikeLj. S tako teorijo lahko sklepamo, da imamo v ljubljanskem sistemu BicikeLj dva tipa postaj. Prvi tip so postaje, locirane v strogem centru, drugi tip pa so postaje okoli centra.

Zaradi odstotkovnega zapisa podobnosti med postajami na lestvici med 0 in 1 je najbolj primerna metrika za ocenjevanje različnosti MAE 4.1. MAE je angleška okrajšava za Mean Absolute Error, slovensko absolutna povprečna napaka. Metrika RMSE, angleško Root Mean Square Error, oziroma slovensko koren povprečne kvadratne napake, bi zaradi zaradi vrednosti med 0 in 1 vračala izjemno majhne vrednosti, ki jih je težje interpretirati.

Čer primerjamo odstotek zasedenosti dveh postaj na minutnem intervalu z metodo povprečne absolutne napake, dobimo povprečno razliko med postajama, podano v odstotkih. Dobljeno vrednost odštejemo od 1 4.2 in tako dobimo vrednost, ki odraža podobnost med postajama.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y_i}| = \frac{1}{n}\sum_{i=1}^{n}|e_i| \tag{4.1}$$

$$Podobnost = 1 - MAE \tag{4.2}$$

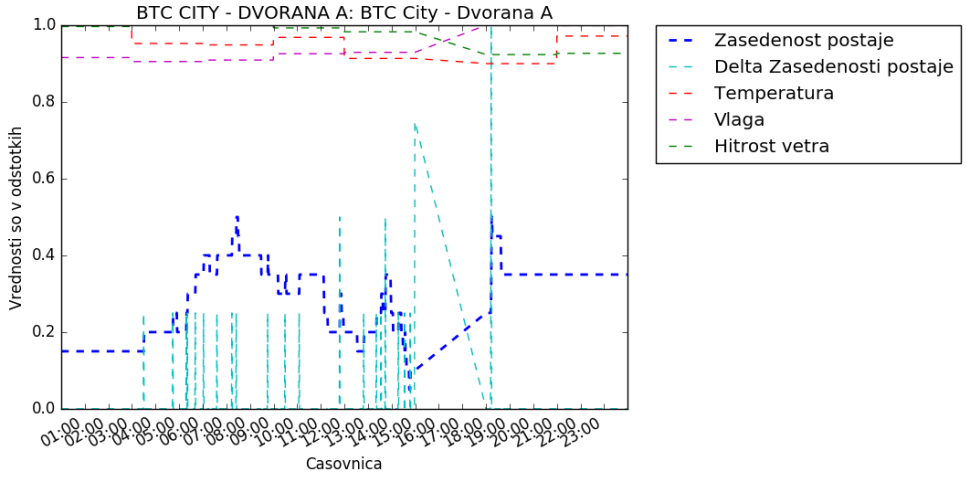
V spodnji tabeli 4.4 si lahko ogledamo deset najbolj podobnih postaj. Povprečna podobnost med vsemi postajami je 0.7095. Opazimo, da so na lestvici večinoma postaje, ki so locirane okoli strogega centra.

Izstopajo pa tudi tri postaje iz nakupovalnega središča BTC. Na prvem mestu sta postaji Citypark in Atlantis, na 6. mestu pa postaji Citypark in BTC dvorana A. Preostala kombinacija med postajo BTC Atlantis in BTC dvorano A pa se ni uvrstila na seznam, saj je na 21. mestu s podobnostjo 0.8143. Glede na to, da imamo med 36 postajami 630 unikatnih parov, jo 21. mesto še vedno uvršča med 4 odstotke najbolj podobnih postaj.

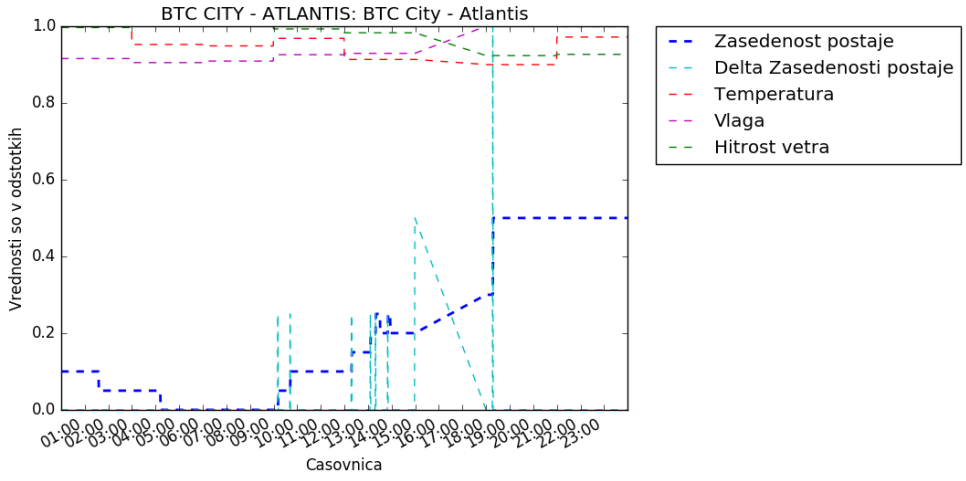
postaja A	postaja B	1 - MAE
CITYPARK	BTC CITY ATLANTIS	0.8457
MIKLOSICEV PARK	BAVARSKI DVOR	0.8451
CANKARJEVA UL.-NAMA	MIKLOSICEV PARK	0.8363
CANKARJEVA UL.-NAMA	BAVARSKI DVOR	0.8321
GRUDNOVO NABREZJE	BAVARSKI DVOR	0.8286
CITYPARK	BTC CITY DVORANA A	0.8275
GRUDNOVO NABREZJE	ILIRSKA ULICA	0.8269
BREG	BAVARSKI DVOR	0.8267
BAVARSKI DVOR	ILIRSKA ULICA	0.8267
KINO SISKA	BARJANSKA C. TRNOVO	0.8231

Tabela 4.4: Lestvica desetih najbolj podobnih postaj.

Med pregledovanjem postaj sem opazil zanimivo lastnost med postajo BTC dvorana A in BTC Atlantis. Postaja BTC dvorana A je veliko bolj aktivna in ima veliko večjo frekvenco izposoj kot postaja pred Atlantisom. Vendar kljub temu lahko med njima najdemo očitne podobnosti. Na spodnjih dveh vizualizacijah 4.6 in 4.7 lahko vidimo vzajemno polnitev postaj, ki se začne okoli 15:00 ure, ter ob 19:00 močno istočasno zapolnitev za približno 20 odstotkov.



Slika 4.6: Vizualizacija ponedeljka, 23. 5. 2016, na postaji BTC dvorana A.



Slika 4.7: Vizualizacija pondedeljka, 23. 5. 2016, na postaji BTC Atlantis.

Tabela 4.5 prikazuje med seboj najmanj podobne si postaje. Najbolj izrazite razlike najdemo med postajami iz strogega centra in dvema postajama, ki sta za Bežigradom.

Postaja na križišču Vojkove in Božičeve ceste je v bližini športnega objekta Stožice, kjer potekajo športni dogodki in tudi koncerti, predstave ter podobni družabni dogodki. Predvidevamo lahko, da je aktivnost postaje močno povezana z dogajanjem na športnem objektu Stožice. Na žalost nimamo podatkov oziroma urnika o tamkajšnjih dogodkih, zato te hipoteze nismo mogli preveriti.

postaja A	postaja B	1 - MAE
BAVARSKI DVOR	VOJKOVA C.-BOZICEVA C.	0.5367
KONGRESNI TRG	VOJKOVA C.-BOZICEVA C.	0.5356
CANKARJEVA UL.-NAMA	DUNAJSKA C.-PS MERCATOR	0.5326
CANKARJEVA UL.-NAMA	VOJKOVA C.-BOZICEVA C.	0.5277
POGACARJEV TRG-TRZNICA	DUNAJSKA C.-PS MERCATOR	0.5236
PRESERNOV TRG	VOJKOVA C.-BOZICEVA C.	0.5184
CANKARJEVA UL.-NAMA	DUNAJSKA C.-PS MERCATOR	0.5326
CANKARJEVA UL.-NAMA	VOJKOVA C.-BOZICEVA C.	0.5277
KONGRESNI TRG	DUNAJSKA C.-PS MERCATOR	0.5159
PRESERNOV TRG	DUNAJSKA C.-PS MERCATOR	0.4879

Tabela 4.5: Lestvica desetih najmanj podobnih postaj.

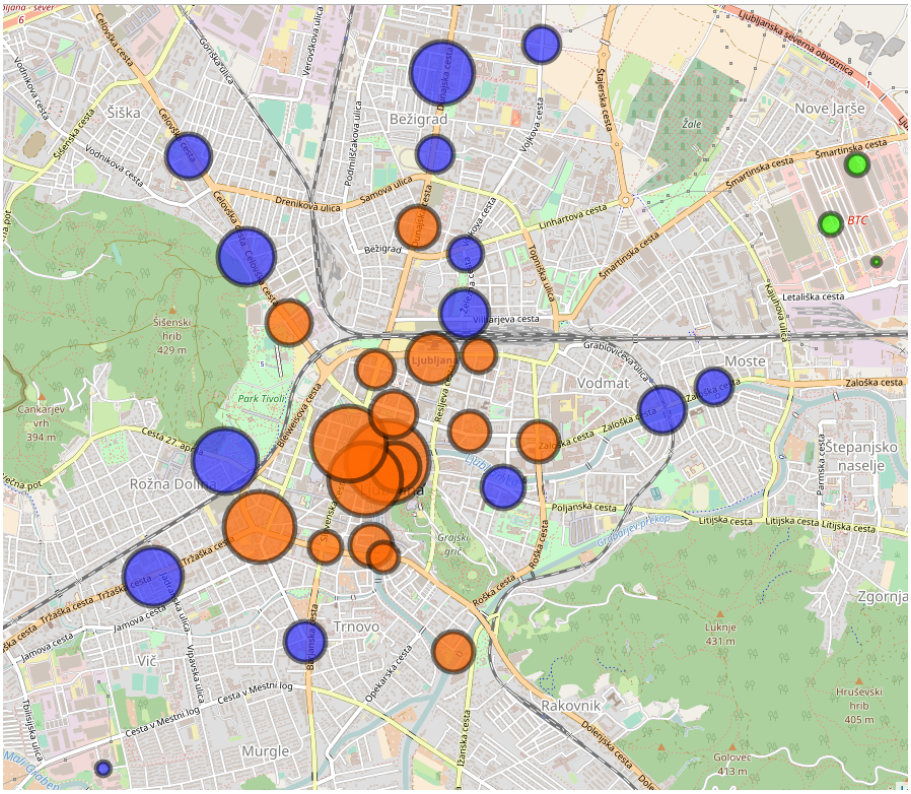
Glede na podatke sem se odločil, da postaje razvrstim v tri kategorije:

- **Kategorija 1**: Postaje v strogem centru oziroma postaje, ki so polne čez dan.
- **Kategorija 2**: Postaje izven strogega centra, ki so nasprotujoče si postajam iz centra, ker pomeni, da so polne ponoči.
- **Kategorija 3**: Postaje znotraj nakupovalnega središča BTC.

Kategorije postaj sem prikazal na interaktivnem zemljevidu 4.8. Vsak krog simbolizira postajo BicikeLj. Velikost kroga predstavlja popularnost postaje oziroma njeno frekvenco izposoj ter vrnitev na postaji. Za izračun radija sem uporabil formulo 4.3. Seštel sem absolutne vrednosti gradienta zasedenosti postaje. Rezultati sem množil z multiplikatorjem "m", v mojem primeru je bil "m"nastavljen na 0.33, tako sem kroge pomanjšal, da je bila njihova velikost bolj primerna za predstavitev na zemljevidu.

$$radij = \left(\sum_{i=1}^{n} |\Delta x_i| \right) * m \tag{4.3}$$

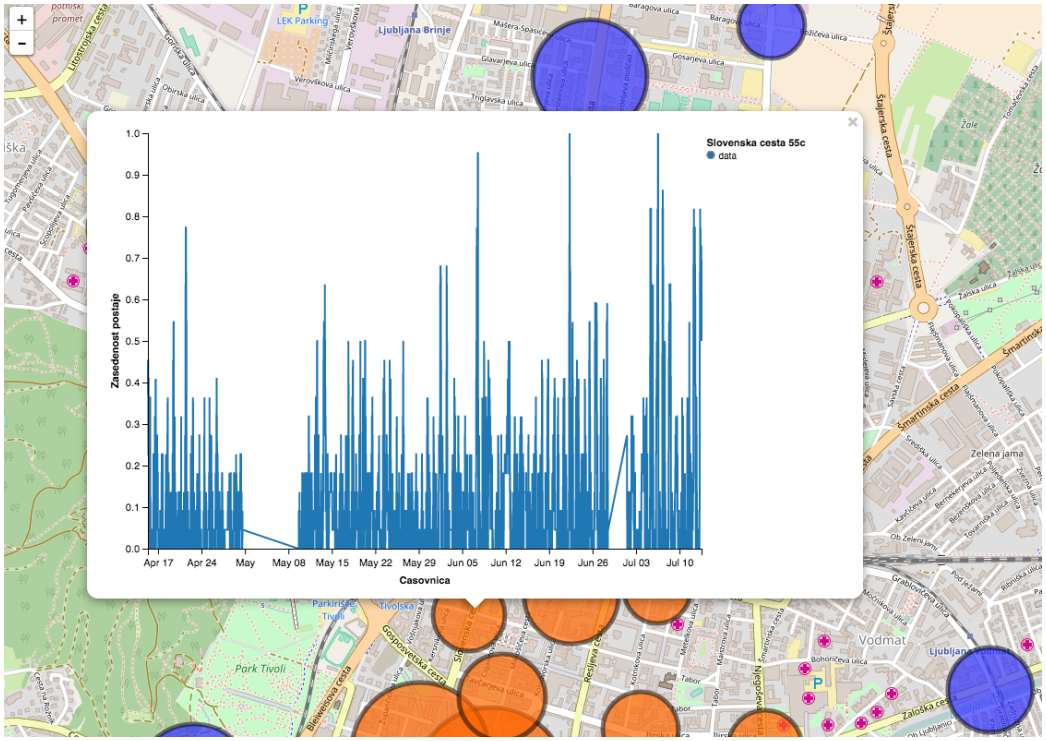
Barva kroga predstavlja kategorijo postaje, kot vidimo, so z oranžno barvo predstavljene postaje v centru, z modro okoliške postaje in z zeleno postaje v nakupovalnem središču BTC.



Slika 4.8: Zaslonski posnetek interaktivnega zemljevida, ki prikazuje kategorije postaj in njihovo popularnost pri strankah.

Če uporabnik interaktivnega zemljevida klikne na krog oziroma marker, ki reprezentatira postajo, se mu odpre okno, v katerem lahko vidi graf in naslov postaje. Graf prikazuje zasedenost postaje skozi čas. Kot vidimo na sliki 4.9, nam za postajo, ki je na Slovenski cesti 55c, manjkajo podatki na začetku meseca maja.

Zavedam se, da graf ni najbolj reprezentativen, vendar sem ga vseeno vključil kot dodatno funkcionalnost zemljevida.



Slika 4.9: Graf markerja.

Poglavje 5

Napovedovanje razpoložljivosti postaj

Za klasifikacijsko napovedovanje zasedenosti postaje BicikeLj sem uporabljal implementacijo odločitvenega drevesa in metodo naključnih dreves, implementiranih v Sparku.

Za regresijsko napovedovanje sem uporabljal algoritem k-NN, vendar ta v Sparku ni podprt, zato sem uporabil implementacijo iz knjižnice Sklearn.

5.1 Odločitveno drevo

Odločitveno drevo [1] je klasifikacijska metoda, ki uporablja podatke za gradnjo tako imenovanega odločitvenega drevesa. Algoritem glede na oceno informativnosti posameznih atributov izbira atribute in ustrezne podmnožice njihovih vrednosti za gradnjo odločitvenih pravil. Dobljene pogoje najpogosteje dodajamo k pogojnemu delu pravila. Sklepni del pravila vsebuje enega ali več razredov, ki jim pripadajo ustrezni učni primeri. Klasifikacija novega primera poteka tako, da se sproži ustrezno pravilo.

5.2 Naključni gozdovi

Metoda naključnih gozdov [1] je namenjena izboljšanju napovedne točnosti drevesnih modelov. Originalno je bila razbita za odločitvena drevesa. Ideja je ustvariti več odločitvenih dreves tako, da se za izbiro najboljšega atributa v vsakem vozlišču naključno izbere majhno število atributov, ki vstopajo v izboru za najboljši atribut. Eno drevo ima en glas pri klasifikaciji novega primera, nov primer klasificiramo kot večinski razred iz vseh ustvarjenih naključnih dreves. Normalno ustvarimo več kot 100 dreves. Učenje takega števila dreves je seveda zahtevno.

5.3 Metoda najbližjih sosedov

Učenje z metodami najbližjih sosedov pomeni shranitev vseh ali pa vsaj podmnožice učnih primerov [1]. Ko izvajamo klasifikacijo ali regresijo, poiščemo podmnožico najbolj podobnih učnih primerov in jih uporabimo za napoved novega primera. Učenja pri teh metodah skoraj ni, zato jim pravimo, da spadajo v kategorijo lenega učenja. Ker večino procesiranja opravimo pri klasifikaciji oziroma regresiji novega primera, je časovna zahtevnost veliko večja kot pri drugih metodah učenja.

5.3.1 K najbližjih sosedov

Najbolj pogosto uporabljena implementacija algoritma se imenuje k najbližjih sosedov, s kratico k-NN (angleško k nearest neighbours). Poznamo regresijsko in klasifikacijsko različico algoritma k-NN. Ko poskusimo klasificirati 5.1 nov primer, poiščemo med učnimi primeri k najbližjih primerov in ga klasificiramo v večinski razred [1].

$$r_x = arg \max_{r \in \{v_1, \ldots, v_{m_0}\}} \sum_{i=1}^{k} \delta(r, r^{(i)}) \tag{5.1}$$

kjer je

$$\sigma(a, b) = \begin{cases} 1, & a = b \\ 0, & a \neq b \end{cases} \tag{5.2}$$

Pri regresiji 5.3 napovemo povprečno vrednost odvisne spremenljivke za k najbližjih sosedov.

$$r_x = \frac{1}{k} \sum_{i=1}^{k} r^{(i)} \tag{5.3}$$

Če imamo opravka z binarno klasifikacijo, parameter k navadno nastavimo na liho število, tako se izognemo neodločeni klasifikaciji v binarnih primerih. Z večanjem parametra k povprečimo napovedi z večjimi bližnjimi primeri, tako se lahko izognemo potencialnemu šumu v podatkih, vendar tudi tvegamo, da k rezultatu prispevajo tudi učni primeri, ki niso dovolj podobni novemu primeru.

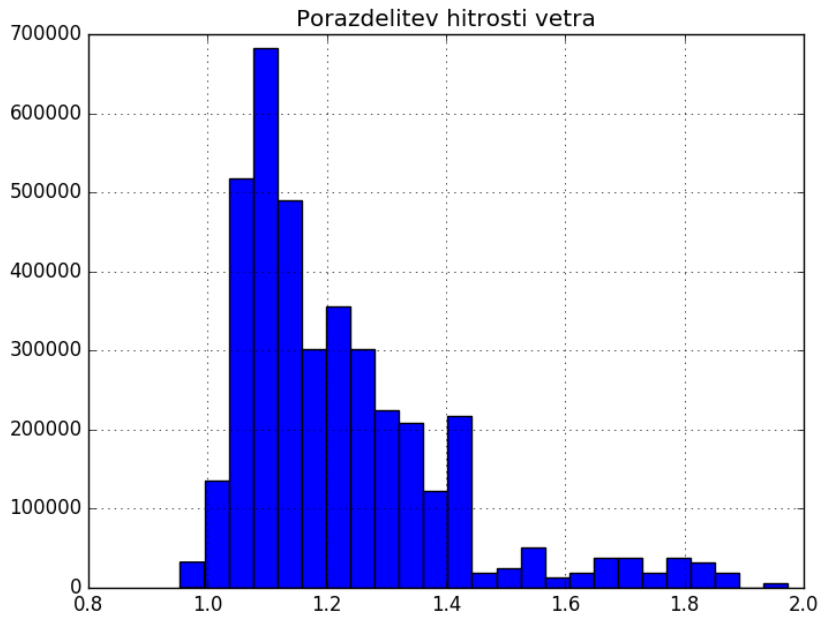
Vse atribute normaliziramo na interval [0, 1]. Algoritem k-NN je občutljiv na izbrano metriko pri računanju razdalje med primeri. Pri zveznih atributih je razdalja enaka absolutni razliki med primeroma, med diskretnimi je razdalja med različnima primerama 1. Pogosto se uporablja oteževanje atributov na razdaljo med dvema primeroma tako, da pomembnejši atributi bolj vplivajo na razdaljo.

5.4 Priprava atributov

Iz večjega števila pregledanih grafov je razvidno, da ima največji vpliv na zasedenost postaje čas. Zanimivo bi bilo videti, ali so dela prosti delavniki podobni nedeljam ali sobotam, vendar v času zajemanja so se vsi prazniki pokrivali z vikendi.

Linuxovo časovno oznako Epoch moramo razdeliti na bolj logične in algoritmu razumljive vrednosti. Ker so vsi podatki iz istega leta ter letnega časa, lahko zanemarimo leto in mesec ter obdržimo informacijo le o dnevu v tednu, uri v dnevu in minuti v uri.

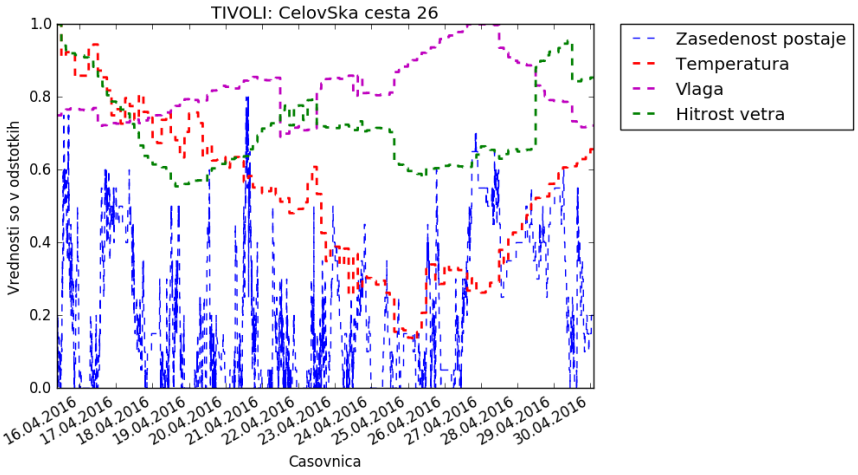
Prav tako nismo bili priča kakršnim koli zahtevnim vremenskim pojavom, kot so nevihte, oziroma vremenu, ki bi ljudi odvrnilo od uporabe sistema. Kot vidimo iz tabele 4.1 in vizualizacije 5.1, je bilo vreme izjemno nevetrovno in umirjeno.



Slika 5.1: Porazdeljenost hitrosti vetra, os X prikazuje povprečno histrost v kilometrih.

Konec aprila smo doživeli sneženje, kar je izjemno nenavaden in redek pojav za ta letni čas. Tudi sam sneg ni odvrnil ljudi od kolesarjenja. Na grafu 5.2 lahko vidimo, da postaja Tivoli ni bila polna ali neaktivna na dan sneženja, 26. 4. 2016.

Opazimo lahko, da je bila vlažnost 100-odstotna okoli 27. in 28. 4. 2016, sklepamo lahko, da sta bila to deževna dneva, a vseeno ni opaziti posebnosti na grafu.



Slika 5.2: Obnašanje postaje Tivoli v času aprilskega sneženja.

Zračni pritisk sem odstranil iz atributov, saj se zdi izjemno neverjetno, da bi se stranka odpovedala uporabi kolesa zaradi trenutnega stanja zračnega pritiska. Prav tako zračnega pritiska nisem vključil v vizualizacije, s čimer sem tako poskusil izboljšati njihovo preglednost.

5.4.1 Testiranje razlike variance

Zaradi preenakomerne porazdeljenosti sem hotel iz atributov odstranil hitrost vetra. Pred končno odločitvijo sem opravil še test razlike variance, implementirane v knjižnici Sklearn. Pri testu definiramo prag ter nato preverimo porazdelitev atributov. Test odstrani vse atribute, katerih porazdelitev ne presežejo zahtevanega praga. Test je pri pragu 0.2 izločil le atribut hitrost vetra.

5.4.2 Diskretizacija atributov

Modele sem poskusil zgraditi z zveznimi in diskretnimi atributi. Z diskretizacijo atributov smo odpravili nepotrebne razlike med določenimi vrednostmi, ki niso pripomogle k izboljšavi napovedi.

- **Dan v tednu**: Glede na grafe so si delavniki ponedeljek, torek, sreda in četrtek podobni med seboj. Petek izstopa z bolj aktivnimi večernimi urami, dopoldan je podoben delavnikom. Sobota in nedelja se tudi razlikujeta od preostalih dni, prav tako tudi med seboj.

Razred	Dan v tednu
0	ponedeljek, torek, sreda, četrtek
1	petek
2	sobota
3	nedelja

Tabela 5.1: Prikaz diskretizacije atributa, ki določa dan v tednu.

- **Ura v dnevu**: Med 00:00 in in 04:00 uro na sistemu načeloma ni mogoče zaznati večjih aktivnosti, zato temu časovnemu intervalu določimo svoj razred. Preostalih 20 ur pa razdelimo v 10 razredov, kjer imajo vsi razredi enako trajanje, tj. 2 uri.

Razred	Ura dneva
0	00:00 - 04:00
1	04:01 - 06:00
3	06:01 - 08:00
...	...
10	22:01 - 23:59

Tabela 5.2: Prikaz diskretizacije ure v dnevu.

- **Minuta v uri**: Minute sem razdelil v tri kategorije, in sicer:

Razred	Minuta ure
0	00-20
1	21-40
2	41-60

Tabela 5.3: Prikaz diskretizacije minute v uri.

Začetek in konec ure se gotovo razlikujeta od intervala okoli središča, saj se navadno ob polni uri začnejo ali končajo aktivnosti ter tako ljudje odhajajo ali prihajajo s svojih aktivnosti.

- **Temperatura**: Čeje temperatura manjša kot 8°, lahko sklepamo, da je hladno za vožnjo s kolesom. V drugem intervalu se nekateri vozijo in nad 15.1° štejemo vreme kot primerno za kolesarjenje za vsakogar.

Razred	Temperatura
0	<8°C
1	8.1°C - 15°C
2	>15.1°C

Tabela 5.4: Prikaz diskretizacije povprečne temperature.

- **Vlaga v zraku**: Vlago smo diskritizirali binarno, in sicer, če je vlaga večja od 90 odstotkov, lahko sklepamo, da zunaj dežuje ali pa je gosta megla.

Razred	Vlaga v zraku
0	0 - 0.9
1	0.91 - 1

Tabela 5.5: Diskretizacije povprečne vlage v zraku.

5.5 Gradnja odločitvenega drevesa

Kot vidimo v spodnji tabeli 5.6, sem zasedenost postaje sem diskritiziral v 4 kategorije. Prvi razred z oznako 0 predstavlja večinski razred s 50 odstotki.

Razred	Zasedenost postaje	Odstotek
0	0 - 0.25	0.50
1	0.25 - 0.5	0.09
2	0.5 - 0.75	0.16
3	0.75 - 1	0.25

Tabela 5.6: Diskretizacija zasedenosti postaje.

Za vsako postajo sem zgradil svoje odločitveno drevo, saj sem tako izboljšal napovedi za približno 3 odstotke. Eksperimentiral sem z različnimi globinami drevesa kot merami za ocenjevanje atributov. Najboljši doseženi rezultat lahko razberemo iz tabele 5.7, dosegel sem ga z drevesom globine 5 in z mero za ocenjevanje Gini-Index.

Tip postaje	Klasifikacijska točnost
Postaje v centru	0.417
Postaje izven centra	0.577
Postaje v BTC-ju	0.447
Povprečje vseh postaj	0.491

Tabela 5.7: Rezultati odločitvenih dreves.

Po pričakovanjih najbolje napovedujemo postaje izven centra in najslabše postaje znotraj centra. Postaje v centru uporabljajo turisti in občasni obiskovalci Ljubljane, katerih vedenja ni mogoče lahko predvideti. Najmanj uspešni smo bili pri postaji pri Bavarskem dvoru, in sicer le v dobrih 20 odstotkih smo jo klasificirali pravilno. Presenetljivo smo najbolje ocenili postajo na Prulah pri Špici, in sicer v 65 odstotkih.

5.6 Gradnja naključnih dreves

Eksperimentalno sem poskusil enake atribute preizkusiti tudi na algoritmu naključnih gozdov. Algoritma nisem mogel uspešno izvršiti pri več kot 10 drevesih in 13 postajah, ki sem jih izbral zaradi prostorske ter časovne zahtevnosti. Postaje sem izbral naključno, in sicer vse 3 postaje iz nakupovalnega središča BTC ter po 5 postaj iz centra in okolice.

Tip postaje	Klasifikacijska točnost
Postaje v centru	0.437
Postaje izven centra	0.558
Postaje v BTC-ju	0.422
Povprečje vseh postaj	0.480

Tabela 5.8: Rezultati naključnih dreves.

Zaradi izredno neobetavnih in slabih rezultatov klasifikacije ter navsezadnje same narave problema sem končal s klasifikacijo in začel regresijski model.

5.7 Gradnja modela k-NN

Z modelom k-NN sem regresijsko napovedoval odstotek polnosti postaje ob podanem času in vremenskih pogojih. Preizkušal sem zgraditi modele z različnimi metrikami razdalje, parametri k. Vse sem preizkušal na zveznih in različno diskretiziranimi atributih.

Tako pri zveznih kot diskretnih atributih se je izkazalo, da z večanjem parametra k drastično izboljšamo napovedi, saj se s povprečenjem med večjim številom primerov znebimo raznih šumov, kot je recimo prihod kombija, ki razporeja kolesa po postajah.

Podatke sem kronološko razdelil na učno in testno množico. Vse vzorce, ki smo jih imeli do 20. 6. 2016 ob 00:00, smo uporabili kot učno množico, preostale podatke pa za testno množico. Iz učne množice smo izvzeli še zadnji teden kot validacijsko množico. Objektivne rezultate smo tako dobili s testiranjem na 23 dneh.

5.7.1 Model k-NN z zveznimi atributi

Atributi, s katerimi je model delal, so bili: dan tedna, ura dneva, minuta ure, temperatura in vlažnost zraka. Vse vrednosti so bile zvezne, uporabil sem Evklidsko razdaljo.

Evklidsko razdaljo 5.4 lahko opišemo kot zračno razdaljo med dvema točkama v prostoru. Izmerimo jo kot dolžino vektorja, ki povezuje obe točki.

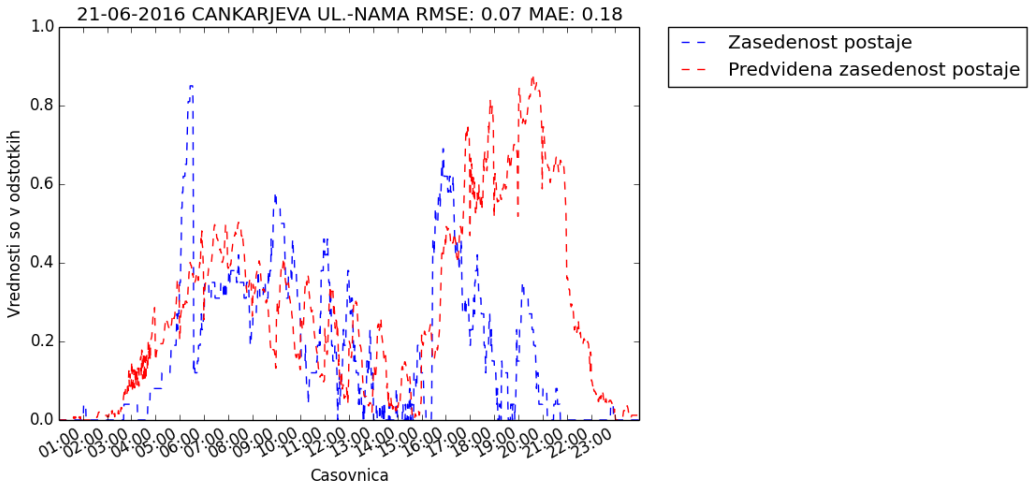
$$d(p, q) = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2} \tag{5.4}$$

Iz tabele 5.9 razberemo, da smo z 10-kratnim povečanjem parametra k izboljšali model za 0.17 po ocenjevalni metriki MAE. Najslabše smo napovedovali polnost postaj, razporejenih okoli centra, najbolje pa smo napovedovali postaje znotraj nakupovalnega središča BTC.

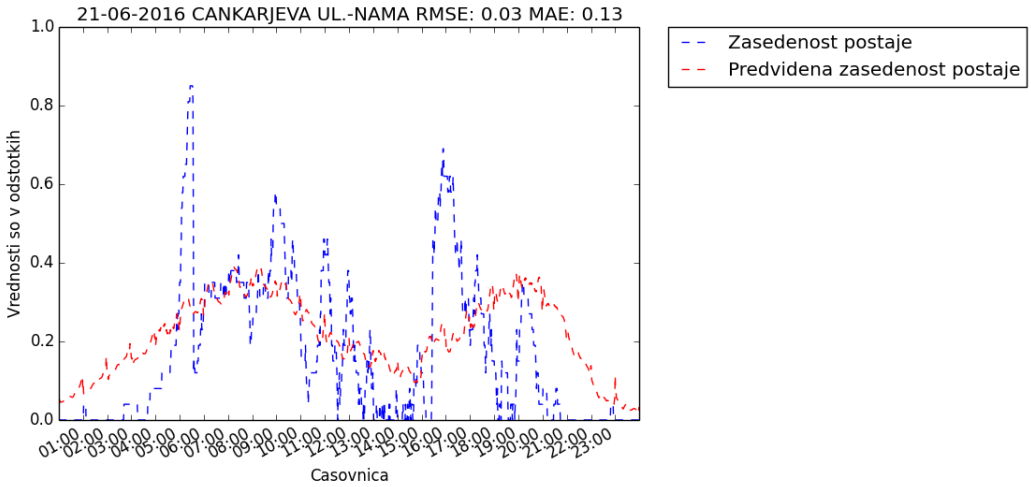
parameter k	tip postaje	MAE
10	center	0.200
	okoli centra	0.213
	BTC	0.159
	povprečje vseh postaj	**0.190**
50	center	0.188
	okoli centra	0.191
	BTC	0.148
	povprečje vseh postaj	**0.175**
100	center	0.190
	okoli centra	0.184
	BTC	0.146
	povprečje vseh postaj	**0.173**

Tabela 5.9: Rezultati pri različnih k parametrih, zveznimi atributih in Evklidski razdalji.

Na spodnjih dveh vizualizacijah 5.3 in 5.4 vidimo grafično predstavitev napovedi s parametroma k 10 in 100. Kot vidimo, je povečanje parametra k izboljšalo napoved za 0.05.



Slika 5.3: Napoved postaje pri Nami s k parametrom 10, zveznimi atributi in Evklidsko razdaljo.



Slika 5.4: Napoved postaje pri Nami s k parametrom 100, zveznimi atributi in Evklidsko razdaljo.

5.7.2 Model k-NN z diskretnimi atributi

Najboljše rezultate dobimo, če atribute diskritiziramo, kot smo to opisali v poglavju 5.4.2, in med njimi merimo razdaljo s Hummingovo metriko 5.5. Hummingova metrika vrne število vseh atributov, ki se razlikujejo, to pa predstavlja tudi našo razdaljo med primeroma.

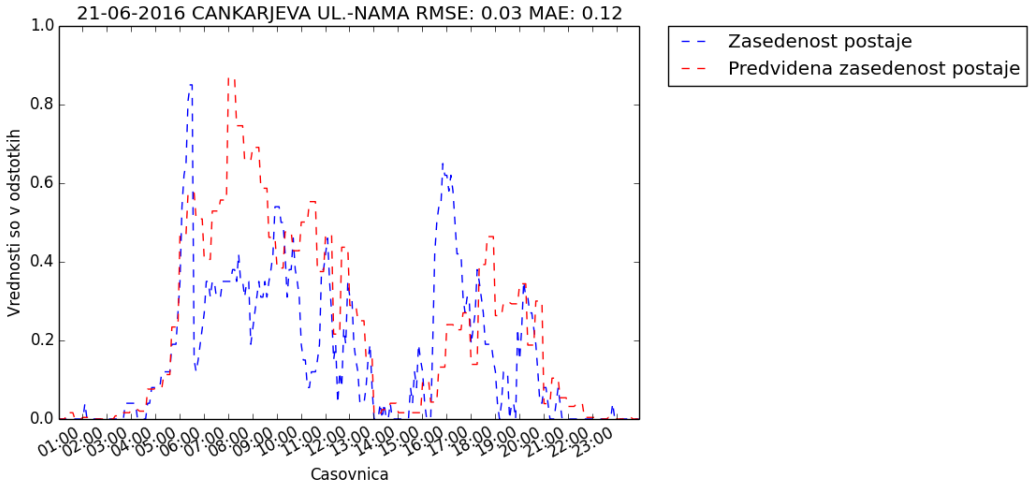
$$d(p, q) = \sum_{i=1}^{n} [q_i \neq p_i] \tag{5.5}$$

Izboljšava najboljšega zveznega rezultata v primerjavi z diskretnim rezultatom je za 0.15 po ocenjevalni matriki MAE.

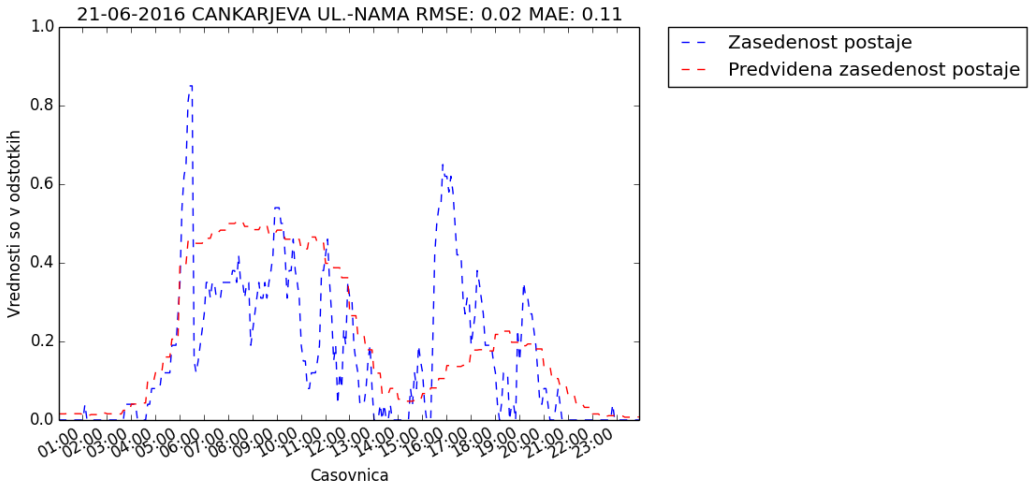
parameter k	tip postaje	MAE
10	center	0.183
	okoli centra	0.188
	BTC	0.144
	povprečje vseh postaj	**0.171**
50	center	0.171
	okoli centra	0.172
	BTC	0.137
	povprečje vseh postaj	**0.160**
100	center	0.170
	okoli centra	0.170
	BTC	0.136
	povprečje vseh postaj	**0.158**

Tabela 5.10: Rezultati pri različnih k parametrih, digitaliziranih atributih in Hummingovi razdalji.

Na spodnjih vizualizacijah 5.5 in 5.6 ter v tabeli 5.10 spet opazimo izboljšanje modela z večanjem parametra k.



Slika 5.5: Napoved postaje pri Nami s k parametrom 10, diskretnimi atributi in Hummingovo razdaljo.



Slika 5.6: Napoved postaje pri Nami s k parametrom 100, diskretnimi atributi in Hummingovo razdaljo.

5.8 Uporabnost napovedovalnega modela

Po izgradnji in optimizaciji napovedovalnih modelov sledi še objektivna ocena njihove uporabne vrednosti. Za ocenjevanje sem zasnoval primitivni napovedovalni model, ki enostavno predvidi, da bo stanje v prihodnosti čez N minut enako trenutnemu stanju na postaji.

Iz razdelka 4.5, kjer opisujemo gradient zasedenosti postaj, lahko vidimo, da je povprečen minutni in petminutni gradient tako majhen, da lahko decimalke, ki se pojavijo, zanemarimo in ga zapišemo kar z 0.

To pomeni, da manjši kot je časovni interval, po katerem primerjamo prihodnje stanje s trenutnim, bolj točen bo primitiven napovedovalni model. Z večanjem časovnega intervala naj bi se tudi povečevala uporabnost modela k-NN. Seveda pa bo velik vpliv igrala tudi popularnost postaje BicikeLj. Večjo frekvenco izposoj in vrnitev koles, kot ju postaja ima, bolj uspešen bi moral biti model k-NN.

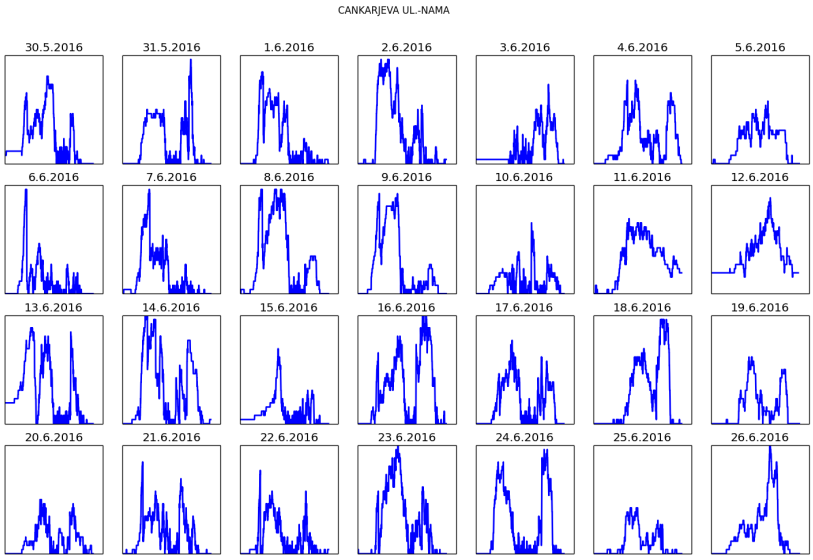
Ob ponovnem pogledu na vizualizacijo 4.8, ki na zemljevidu prikazuje tipe postaj in frekvenco njihove uporabe, vidimo, da lahko pričakujemo veliko zgodnejšo uporabno vrednost modela k-NN pri postajah, ki so v strogem centru in smo jih uvrstili kot postaje tipa 1.

V prvem stolpcu tabele 5.11 lahko preberemo ločljivo minuto, ki na dani postaji označuje čas v prihodnosti, po katerem je bolj uspešen model k-NN. Po pričakovanjih je model na postajah iz centra, ki imajo večjo frekvenco delovanja, veliko bolj uspešen kot na bolj umirjenih postajah izven centra.

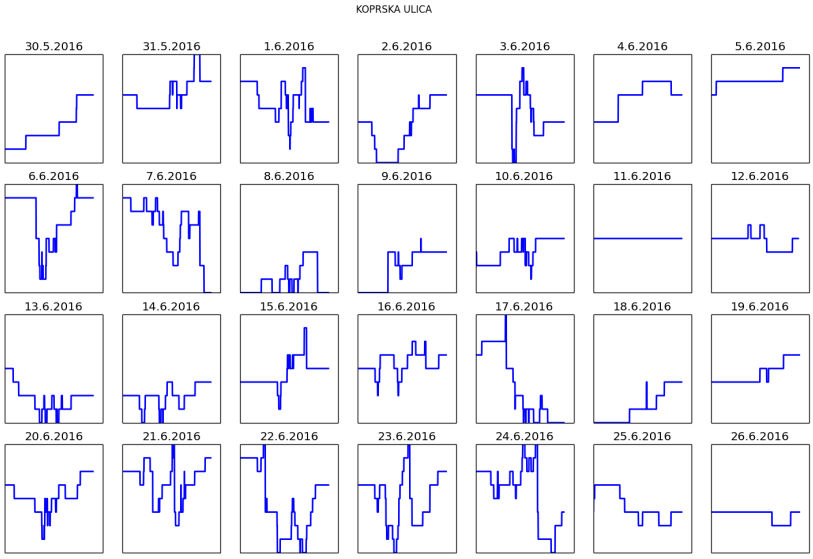
minuta	postaja	tip	zamik MAE	model MAE
65	PRESERNOV TRG	center	0.177	0.171
65	CANKARJEVA UL.-NAMA	center	0.139	0.136
75	KONGRESNI TRG	center	0.173	0.172
120	KINO SISKA	okolica	0.152	0.150
125	C. STAREJSIH TRNOVO	okolica	0.151	0.151
155	TRZNICA MOSTE	okolica	0.193	0.190
185	CITYPARK	BTC	0.131	0.131
200	BREG	center	0.190	0.190
260	BTC CITY - DVORANA A	BTC	0.161	0.159
415	KOPRSKA ULICA	okolica	0.207	0.206
1680	BTC CITY - ATLANTIS	BTC	0.120	0.120

Tabela 5.11: Pregled uporabnosti modelov k-NN za nekaj izbranih postaj.

Osebno mislim, da je časovni interval, pri katerem model k-NN začne premagovati primitivni model pri določenih postajah, izjemno velik. Presenetila me je postaja pri Atlantisu, kjer moramo za uspešnejšo uporabo modela k-NN napovedovati več kot 28 ur v prihodnost.



Slika 5.7: Zasedenosti postaje na Cankarjevi ulici pri Nami v štirih zaporednih tednih.



Slika 5.8: Zasedenosti postaje na Koprski ulici v štirih zaporednih tednih.

Na vizualizaciji 4.3, ki prikazuje 4-tedenski pregled delovanja postaje na Prešernovem trgu, vidimo veliko vsakodnevno aktivnost postaje. Prav tako smo na dnevni vizualizaciji 4.7 prikazali enodnevno delovanje postaje pri Atlantisu, kjer je k-NN model najbolj neuspešen zaradi nizke frekvence vrnitev in izposoj koles.

Ker sem v besedilo diplomske naloge že vključil vizualizacije teh dveh postaj, sem se odločil, da na prejšnji strani prikažem postajo s Cankarjeve ulice pri Nami 5.7 ter postajo na Koprski ulici 5.8. Tako lahko na teh dveh vizualizacijah opazimo kontrast med številom aktivnosti na postajah Nama in Koprska ulica.

Razlika med številom dogodkov oziroma frekvenco izposoj in vrnitev kolos, ki so se pripetili na vsaki postaji, pa opravičuje dolžino časovnega intervala, ki ga potrebujemo za uspešno napovedovanje z modelom k-NN.

Na postaji pri Koprski ulici lahko celo zasledimo obdobje od konca dneva 10. 6. 2016 do približno sredine dneva 12. 6. 2016, ko postaja ni imeli niti ene same izposoje ali vrnitve kolosa. MAE-napaka primitivnega modela ob takem pojavu bi bila 0, omenjeno obdobje je izjemno redko in težko predvidljivo z modelom k-NN ali s katero drugo napovedovalno metodo. Tak pojav je skoraj nemogoč na popularnejših postajah.

Poglavje 6

Sklepne ugotovitve

V okviru diplomske naloge sem razvil program, ki je pridobival podatke iz javnih API-jev in jih shranjeval v podatkovno bazo. Kronološko delovanja programa sem zagotovili z Linuxovim orodjem Cron.

Postavil sem računalniško gručo, ki je shajala iz dveh najbolj aktualnih in naprednih tehnologij za shranjevanje ter obdelovanje podatkov. Hadoop in Spark sta izjemno močni orodji, zavedam se, da nisem niti približno mogel izrabil njunega potenciala, vendar želel sem se spoznati z omenjenimi tehnologijami ter osnovnimi koncepti delovanja računalniške gruče. Sama postavitev sistema je zahtevala različna znanja od konfiguracije računalniških omrežij do poznavanja Linuxove strukture in njegove ukazne lupine.

V zadnjem delu naloge sem obdeloval zbrane podatke, na njih poskusil poiskati razne zanimivosti in zgraditi model za napovedovanje zasedenost postaj BicikeLj. Po izgradnji in optimizaciji napovedovalnega modela sem ga ocenil ter preveril njegovo praktično uporabnost.

Ugibam, da bi se dalo samo natančnost modela močno izboljšati, če bi imeli še podatke oziroma urnike raznih dogodkov, ki se odvijajo po Ljubljani. Sistem bi lahko tudi objavili v obliki spletne aplikacije, kjer bi lahko uporabniki imeli vpogled v napovedi.

Kot rednemu uporabniku sistema BicikeLj bi lahko bilo v določenih situacijah praktično vedeti, ali lahko računam na izposojo kolesa ob določenem času. Vseeno močno dvomim, da povprečen uporabnik sistema BicikeLj vnaprej predvideva svoje vožnje s kolesom, saj jih niti sam ne. Razlog za to je, da ima sam sistem BicikeLj izredno dobre alternative, kot so na primer zasebno kolo, javni prevoz ali kar pešačenje.

Delam v podjetju Comtrade na oddelku za mobilnost in turizem, kjer aktivno razvijamo mobilnostno platformo. Platforma je trenutno uporabljena tudi v produkciji, in sicer za potrebe ljubljanskega sistema za souporabo vozil, imenovanega Avant2Go.

Ko sem začel delati na diplomski nalogi, je bil projekt še v začetni fazi in ni imeli niti dovolj aktivnih uporabnikov niti izposoj, da bi lahko zbral dovolj podatkov za učinkovito podatkovno rudarjenje. Zato sem se odločil za izdelavo diplomske naloge na že uveljavljenem sistemu BicikeLj.

Sistema sta si v nekaterih pogledih izjemno podobno, saj oba svojim klientom zagotavljata mobilnost. Glavna razlika, ki bi jo izpostavil, je to, da uporabniki sistema za izposojo avtomobilov svoje vožnje in uporabo sistema načrtujejo veliko prej kot uporabniki sistema BicikeLj, saj avtomobile lahko uporabljajo za prihod na letališče in druge bolj oddaljene lokacije.

Model za napovedovanje polnosti postaj za sistem izposoje avtomobilov bi tako imel veliko večjo praktično uporabnost.

Literatura

[1] I. Kononenko, M. R. Šikonja. *Inteligentni sistemi.*
Založba FE in FRI, 2010.

[2] K. Sitto, M. Presser. *Field guide to Hadoop.*
O'Reilly Media, 2015.

[3] M. Frampton. *Mastering Apache Spark.*
Packt Publishing, 2015.

[4] H. Karau. *Fast Data Processing with Spark.*
Packt Publishing, 2013.

[5] K. Sankar, H. Karau. *Fast Data Processing with Second Edition.*
Packt Publishing, 2015.

[6] JCDecaux Developer [Online].
Dosegljivo: https://developer.jcdecaux.com [Dostopano 5.4.2016].

[7] OpenWeatherMap, Inc. [Online].
Dosegljivo: https://openweathermap.org/api [Dostopano 5.4.2016].

[8] NodeJS [Online].
Dosegljivo: https://nodejs.org/dist/latest-v6.x/docs/api/
[Dostopano 8.4.2016]

[9] Slack Technologies [Online].
Dosegljivo: https://slack.com/is [Dostopano 10.4.2016].

[10] MongoDB, Inc. [Online].
Dosegljivo: https://www.mongodb.com/ [Dostopano 1.4.2016].

[11] Async [Online].
Dosegljivo: https://github.com/caolan/async [Dostopano 5.4.2016].

[12] Mongoose [Online].
Dosegljivo: https://github.com/Automattic/mongoose
[Dostopano 5.4.2016].

[13] Pedregosa, F. and Varoquaux, G. and Gramfort, A. and Michel, V. and Thirion, B. and Grisel, O. and Blondel, M. and Prettenhofer, P. and Weiss, R. and Dubourg, V. and Vanderplas, J. and Passos, A. and Cournapeau, D. and Brucher, M. and Perrot, M. and Duchesnay, E., "Scikit-learn: Machine Learning in Python", *Journal of Machine Learning Research*, št. 12, str. 2825–2830, 2011.

[14] Request [Online].
Dosegljivo: https://github.com/request/request [Dostopano 5.4.2016].

[15] Winston [Online].
Dosegljivo: https://github.com/winstonjs/winston
[Dostopano 5.4.2016].

[16] Winston-Slackbotuser [Online].
Dosegljivo: https://github.com/candrholdings/winston-slackbotuser
[Dostopano 5.4.2016].

[17] Apache Spark [Online].
Dosegljivo: http://spark.apache.org/docs/2.1.0/
[Dostopano 28.12.2016].

[18] Apache Spark [Online].
Dosegljivo: http://www.jmlr.org/papers/volume17/15-237/15-237.pdf/
[Dostopano 5.1.2016].

[19] Apache Spark MLib [Online].
Dosegljivo: http://hadoop.apache.org/docs/r2.7.3/
[Dostopano 1.9.2016].

[20] Cron [Online].
Dosegljivo: http://www.unix.com/man-page/posix/1posix/crontab
[Dostopano 3.4.2016].

[21] Python2.7 [Online].
Dosegljivo: https://docs.python.org/2/ [Dostopano 10.10.2016].

[22] Numpy [Online].
Dosegljivo: http://www.numpy.org/ [Dostopano 15.10.2016].

[23] Pandas [Online].
Dosegljivo: http://pandas.pydata.org/ [Dostopano 15.10.2016].

[24] Matplotlib [Online].
Dosegljivo: http://matplotlib.org/ [Dostopano 2.2.2017].

[25] Folium [Online].
Dosegljivo: https://github.com/python-visualization/folium
[Dostopano 28.1.2017].

[26] Pymongo [Online].
Dosegljivo: https://api.mongodb.com/python/current/
[Dostopano 10.10.2016].

[27] Linux Network Administrators Guide [Online].
Dosegljivo: http://www.tldp.org/LDP/nag2/nag2.pdf
[Dostopano 27.8.2016]

[28] Doug Cutting Wikipedia [Online]
Dosegljivo: https://en.wikipedia.org/wiki/Doug_Cutting
[Dostopano 4.2.2017]

[29] Matei Zaharia Wikipedia [Online]
Dosegljivo: https://en.wikipedia.org/wiki/Matei_Zaharia
[Dostopano 4.2.2017]

[30] Biggest Hadoop Cluster [Online]
Dosegljivo: https://wiki.apache.org/hadoop/PoweredBy#Y
[Dostopano 4.2.2017]

[31] [Online] Apache Spark officially sets a new record in large-scale sorting
Dosegljivo: https://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html
[Dostopano 4.2.2017]